§8 ПІДПРОГРАМИ

Підпрограми

Підпрограма – це логічно незалежний фрагмент коду, оформлений спеціальним чином за правилами мови програмування, до якого можна (багаторазово) звернутися з будь-якого місця програми.

Підпрограми дозволяють значно скоротити об'єм коду та спросити його розуміння.

Звернення до підпрограми називають **викликом підпрограми**.

Підпрограми бувають **іменованими** та **анонімними**

У загальному випадку у програмуванні підпрограми поділяють на **процедури** та **функції**.

Процедура – це фрагмент коду (послідовність інструкцій), призначений для розв'язання певної задачі.

Функція – процедура, що повертає у місце виклику певне значення, що є результатом її виконання.

У Python усі підпрограми є функціями.

З багатьма функціями ми вже зустрічалися (тобто використовували) при написанні програм раніше, наприклад

```
len(l)
max(l)
```

Це були вбудовані функції. Давайте познайомимося з тим, як створювати власні функції у Python, правилами виклику функцій та іншими особливостями роботи з функціями.

Опис функції

Перш ніж використовувати функцію її необхідно створити або, кажучи термінами програмування, **описати**.

Для опису функції у Python необхідно:

- задати ім'я функції (для іменованих функцій);
- описати аргументи функції (не обов'язково);
- описати програмний код функції;
- описати повернення результату (не обов'язково).

Отже, як видно з вищенаведеного, єдиною обов'язковою дією для опису функції у Python є опис програмного коду функції. Розглянемо синтаксис іменованих функцій (з анонімними функціями познайомимося пізніше)

```
def function_name(args):
    function_body
```

тут **def** – ключове слово, що вказує на початок опису функції, `function_name` – ім'я функції, що задається програмістом. `args` – послідовність аргументів (параметрів) функції, `function_body` – тіло функції, тобто програмний код функції.

Якщо опис функції не містить жодного аргументу, то `args` опускають. При цьому круглі дужки **не опускаються**:

```
def function_name():
    function_body
```

Для повернення результату використовується оператор

```
return result
```

де `result` – результат виконання функції. Інструкція **return** завершує виконання функції, навіть якщо за нею слідують інші інструкції. Тому її часто використовують у якості оператора примусового завершення функції, наприклад разом з умовним оператором або в циклах.

Зауважимо, що функція у Python може повертати будь-який об'єкт – число, рядок, список, кортеж, тощо. Якщо тіло функції не містить інструкції **return** або під час виклику функції інтерпретатор не натрапляє на неї, то функція повертає значення **None** ("ніщо").

Опишемо для прикладу функцію, що визначає більше з двох чисел:

```
def max2(a, b):
    if a > b:
        return a
    else:
        return b
```

Отже, ім'я цієї функції `max2`. Вона має два аргументи `a`, `b` та повертає значення `a`, якщо `a > b` і значення `b` у іншому випадку.

Тепер, після опису, функцію `max2` можна викликати:

```
>>> max2(2, 3)
3
```

```
>>> m = max2(444, 555)
>>> print(m)
555
```

Результат виклику функції можна використовувати як аргументи для виклику функції. Наприклад, описану вище функцію можна використовувати для знаходження найбільшого серед трьох чисел

```
>>> max2(11, max2(12, 13))
13
```

Повернення кількох значень функцією

Як було сказано раніше, за допомогою інструкції **return** можна повертати будь-який об'єкт Python, розпакований кортеж у тому ж числі. Цей механізм використовується для реалізації випадку повернення кількох значень однією функцією. Для цього необхідно у інструкції **return** через кому перерахувати значення, які повертає функція.

```
def func(...):
    ...
    return result1, result2, ..., resultN
```

Фактично така функція повертає кортеж

```
(result1, result2, ..., resultN)
```

При цьому його можна зразу розпакувати у послідовність змінних (кількість яких повинна дорівнювати довжині кортежу)

```
re1, res2, ..., resN = func(...)
```

Приклад 8.1. Визначити символ, що входить до рядка найбільшу кількість разів.

Розв'язок. Розіб'ємо задачу на дві частини:

1. Побудова на базі рядка словника, у якому ключами будуть символи, а значеннями кількість входжень відповідного символа до рядка.
2. Пошук у словнику ключа якому відповідає найбільше значення.

Оформимо кожну з частин у вигляді функції.

Перша функція `construct_dict` буде у якості параметра приймати рядок `S`, і повертати словник `char_dict` у якості результату.

```
def construct_dict(S):  # S - формальний параметр (рядок)
    char_dict = {}      # char_dict - словник символів
    for c in S:
        if c not in char_dict:
            char_dict[c] = S.count(c)
    return char_dict
```

Можемо перевірити роботу функції, викликавши її для деякого рядку, наприклад

```
>>> D = construct_dict("молоко")
>>> print(D)
```

Результат буде таким

```
{'л': 1, 'о': 3, 'к': 1, 'м': 1}
```

Друга функція `find_most_popular_char` буде приймати побудований словник і повертати пару значень – найпопулярніший символ і кількість його входжень

```
def find_most_popular_char(D):
    max_char = ''  # найпопулярніший символ
    max_num = 0    # кількість його входжень
    for c, num in D.items():
        if num > max_num:
            max_char = c
            max_num = num
    return max_char, max_num # функція повертає пару
                             # значень max_char, max_num
```

Для перевірити коректності роботи функції, здійснимо її виклик для раніше побудованого словника

```
>>> D = construct_dict("молоко")
>>> C, N = find_most_popular_char(D)
>>> print(C, N)
о 3
```

Перевіривши правильність роботи обох функцій, можемо написати головну програму. Отже, остаточно програма, що виконує поставлене у задачі завдання буде мати вигляд

```
def construct_dict(S):
    ...

def find_most_popular_char(D):
    ...

S = input("Задайте рядок ")
D = construct_dict(S)
C, N = find_most_popular_char(D)
print("символ '%s' входить у рядок %d разів" % (C, N))
```

Тут символами `...` умовно позначаються раніше наведені тіла функцій.

Аргументи функції

Аргументи функції (їх також називають **параметрами**), це вхідні параметри функції, які передаються функції на обробку. Від них залежить не лише поведінка функції, а й результат її виконання.

Аргументи функції розділяють на формальні параметри та фактичні аргументи. Формальні параметри задаються під час опису функції. Формальні параметри не мають конкретних значень – вони лише використовуються для опису шаблону функції. Наприклад, у вищенаведеному описі функції

```
def max2(a, b):
    ...
```

`a, b` є формальними параметрам.

Фактичні аргументи вказуються **під час виклику функції**. Фактичні аргументи мають конкретні значення (літерал, значення виразу, об'єкт). Наприклад, під час виклику функції

```
>>> max2(2, 3)
3
```

2 і 3 є фактичними аргументами.

Передача параметрів у функцію

Фактично будь-яка функція є шаблоном коду, що залежить від формальних параметрів. Щоб виконати функцію її потрібно викликати з фактичними аргументами. При цьому відбувається **передача параметрів у функцію**.

Розберемо передачу параметрів у функцію на прикладі. Нехай описано функцію

```
def func(param1, param2, param3):
    ...
```

що має три формальних параметри `param1, param2, param3`.

Під час виклику функції з фактичними аргументами `arg1, arg2, arg3`

```
func(arg1, arg2, arg3)
```

фактичні параметри підставляються в шаблон функції замість формальних, тобто відбувається таке присвоєння

```
param1 = arg1
param2 = arg2
param3 = arg3
```

після чого відбувається виконання тіла функції. Це присвоєння і буде передачею параметрів у функцію.

При цьому потрібно пам'ятати, що операція присвоювання копіює не значення змінних (або об'єктів), а їхні адреси (посилання) у пам'яті. Тобто, після вищенаведеного присвоєння Змінні `param1` та `arg1` будуть посилатися на один і той же об'єкт в пам'яті. Аналогічна ситуація буде і з іншими парами змінних `param2` та `arg2` і `param3` та `arg3`. Використовуючи термінологію програмування, кажуть, що передача параметрів у Python відбувається **за посиланням**, а не **за значенням** (як наприклад, у мовах програмування Pascal або C\C++).

Передача параметрів за посиланням вимагає від програміста більше обережності при написанні функцій, Наприклад, при зміні значень формальних параметрів в тілі функції, що належать до незмінюваних об'єктів, фактичні аргументи не зміняться, а от зміни над формальними аргументами, що є змінюваними відобразяться на зміні фактичних параметрів.

Щоб краще зрозуміти вищенаведене, наведемо приклади.

Опишемо функцію, що збільшує значення аргументу на 1.

```
def increment_Int(a):
    a += 1      # збільшуємо значення змінної a на 1
```

Будемо викликати функцію для змінної `myInt` зі значенням 1. Після чого виведемо значення змінної `myInt` на екран

```
myInt = 1
increment_Int(myInt)
print(myInt)
```

Результатом буде

```
1
```

Спробуємо розібратися чому значення змінної `myInt` не змінилося, якщо у тілі функції ми збільшили `a`, а значить і `myInt` на 1.

В момент виклику функції `increment_Int` обидві змінні – і формальний параметр `a` і фактичний аргумент `myInt` посилаються на один об'єкт в пам'яті, що належить до цілого типу. Після виконання інструкції `a += 1` (оскільки `a` належить до `immutable` типу), формальний параметр буде посилатися вже на інший об'єкт цілого типу зі значенням **2**, а фактичний аргумент залишиться незмінним.

Розглянемо інший приклад, у якому фактичний аргумент буде належати до змінюваного типу. Опишемо функцію, що додає до списку число **1**.

```
def increment_list(a):
    a.append(1)    # додаємо до списку a число 1

myList = [1]
increment_list(myList)
print(myList)
```

результатом виконання вищенаведеного коду буде

```
[1, 1]
```

Аналогічно попередньому прикладу, в момент виклику функції `increment_list` обидві змінні – і формальний параметр `a` і фактичний аргумент `myList` посилаються на один і той же список. Після виконання інструкції `a.append(1)` (оскільки `a` належить до змінюваного типу), формальний параметр `a`, а разом з ним і фактичний аргумент `myList` доповняться елементом **1**.

Типові значення аргументів

Для формальних параметрів функцій часто задають **типові значення** (eng: "by default", рус: "по умолчанию"). Типові значення функції, це такі значення, які будуть використовуватися функцією, якщо аргумент функції явно не заданий під час виклику цієї функції.

Для того, щоб задати типове значення формальному параметру, його задають в описі функції через оператор присвоєння, наприклад:

```
def func(param = "Типове значення"):
    print(param)
```

Здійснимо виклик описаної функцію з заданим фактичним аргументом `"Моє значення"` та з використанням типового значення.

```
>>> func("Моє значення")
Моє значення
>>> func()
Типове значення
```

Як ми знаємо, функція може мати довільну кількість аргументів. Частина з цих аргументів може мати типові значення, а інша частина – ні. Існує важливе правило для використання типових значень: У описі функції, у переліку формальних параметрів спочатку задаються параметри, що не мають типових значень, а аргументи, що мають типові значення задаються у описі функції в кінці.

```
def func(p1, p2, p3 = u, p4 = v):
    ...
```

Під час виклику функції, типові значення задаються відповідно до порядку у списку аргументів. Наприклад, якщо для вищенаведеної функції виклик буде мати вигляд

```
func(a1, a2, a)
```

то це означатиме, що для параметру `p4` використовується типове значення `v`, а для `p3` задане значення `a`.

Приклад 8.2. 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. Поставлена задача була розв'язана у прикладі 3.26. Оформимо код наближеного обчислення у вигляді функції.

Як було встановлено раніше, функція e^x визначається як границя послідовності S_n, що задана системою рекурентних співвідношень

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

При цьому під наближеним (з точністю ε) значенням границі послідовності S_n будемо розуміти такий член S_N , що виконується співвідношення

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

Отже, опишемо функцію, що буде мати два формальних параметри `x`, `eps`, що відповідають x і ε, причому формальний параметр `eps` буде мати типове значення `0.0001`.

```
# Описуємо функцію exp
def exp(x, eps = 0.0001):
    S = a = 1
    n = 0
    while abs(a) >= eps:
        n += 1
        a *= x / float(n)
        S += a
    return S

# головна програма
x = float(input("x = "))
y = exp(x) # використовуємо типове значення параметра eps
print ("exp(%f) = %f" % (x, y))
```

Позиційні та ключові параметри

Аргументи функцій, розглянуті раніше, належать до **позиційних параметрів**, тому що співставлення між фактичними аргументами та формальними параметрами здійснюється за позицією у списку параметрів.

Проте, таке співставлення можна здійснювати не за позицією, а за відповідним іменем формального параметра. Наприклад, для функції

```
def func(param1, param2, param3):
    ...
```

виклик може мати вигляд

```
y = func(param1 = a1, param2 = a2, param3 = a3)
```

Такі фактичні аргументи (`a1, a2, a3`) називаються **ключовими**, оскільки передача аргументів функції здійснюється парами – ключ (ім'я формального параметра)-значення (ім'я фактичного аргументу).

Основна перевага ключових параметрів у тому, що їхній порядок розміщення під час виклику функції не має значення. Наприклад, виклик вищенаведеної функції може виглядати і таким чином

```
y = func(param2 = a2, param3 = a3, param1 = a1)
```

Функції з довільною кількістю параметрів

У Python можна описувати функції, що приймають довільну кількість параметрів, причому, як позиційних, так і ключових.

Для довільної кількості позиційних параметрів використовують оператор `*` перед іменем аргументу.

```
def func(*args):
    ...
```

Під час виклику такої функції аргументи передаються звичайним чином – записуються через кому:

```
x = func()            # виклик функції без аргументів
func(x, y)            # виклик функції з двома аргументами
func(a1, a2, a3, a4) # виклик функції з чотирма аргументами
```

Параметр `args` (без `*`) при такому описі функції є кортежом. Переконатися у цьому можна на такому простому прикладі

```
>>> def func(*args):
        print(args)

>>> func(1, 2.0, "string")
(1, 2.0, 'string')
```

Приклад 8.3. Опишемо функцію, що знаходить суму елементів заданої послідовності.

Розв'язок. Оскільки не відомо, скільки аргументів буде мати функція, опишемо її як таку, що приймає довільну кількість позиційних аргументів.

```
def suma(*elements):
    res = 0
    # Оскільки elements – кортеж скористаємося циклом for
```

```
    for el in elements:
        res += el
    return res

# Головна програма полягає у виклиці функції suma.
print(suma(1, 2, 3, 4, 5))
```

Для довільної кількості ключових параметрів використовують оператор `**` перед іменем аргументу.

```
def func(**kwargs):
    ...
```

Параметр `kwargs` при цьому є словником у якому формальні параметри є ключами, а фактичні – значеннями.

```
>>> def func1(**kwargs):
        print(kwargs)

>>> func1(x = 1, y = 2, z = 3)
{'x': 1, 'z': 3, 'y': 2}
```

Зауважимо, що в Python можна описувати функції, що приймають довільну кількість позиційних і ключових параметрів. Стандартний опис тоді такої функції матиме вигляд

```
def func(*args, **kwargs):
    ...
```

Локальні та глобальні змінні

При описі функцій у тілі функції ми часто використовували змінні, що не були формальними параметрами. Наприклад, у описі функції

```
def suma(*elements):
    res = 0
    for el in elements:
        res += el
    return res
```

такими змінними є `res` та `el`.

Такі змінні, які визначені в тілі функції називаються **локальними** і є доступними лише в тілі функції у якій визначені. Щоб переконатися у цьому опишемо функцію, що має локальну змінну та спробуємо скористатися цією змінною поза межами тіла функції

```
def func():
    s = 1  # s - локальна змінна

func()
print(s)
```

Результатом виконання вищенаведеного коду буде повідомлення про помилку, у якому буде сказано, що змінна `s` не є визначеною:

```
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    s
NameError: name 's' is not defined
```

Змінні, що є досяжними поза функцією, називаються **глобальними**.

```
x = 1          # x - глобальна змінна

def func():
    print(x) # Виводимо на екран глобальну змінну x

func()         # Викликаємо вищеописану функцію
```

Результатом виконання вищенаведеного коду буде

```
1
```

Не розуміння різниці між локальними і глобальними змінними часто призводить до плутанини і, відповідно, до помилок у коді.

Щоб краще засвоїти різницю між локальними та глобальними змінними, розглянемо кілька прикладів.

```
x = 1          # x - глобальна змінна зі значенням 1

def func():
    print(x) # Виводимо на екран глобальну змінну x
```

```
func()          # Викликаємо вищеописану функцію,
                # що виводить на екран x
print(x)        # Виводимо на екран глобальну змінну x
```

Результатом виконання цього коду буде

```
1
1
```

Виконаємо вищенаведену програму, модифікувавши функцію `func`.

```
x = 1           # x - глобальна змінна зі значенням 1

def func():
    x = 222     # Змінимо значення x
    print(x)    # Виводимо на екран глобальну змінну x

func()          # Викликаємо вищеописану функцію,
                # що виводить на екран x
print(x)        # Виводимо на екран глобальну змінну x
```

Результатом виконання цього коду буде

```
222
1
```

Такий результат пояснюється тим, що у першому випадку, інтерпретатор, не знайшовши у функції явного опису змінної `x`, зрозумів, що треба використовувати глобальну змінну `x`. У другому випадку, операцію присвоєння

```
    x = 222     # Змінимо значення x
```

інтерпретатор сприйняв не як модифікацію глобальної змінної, а як створення нової змінної. Глобальна ж змінна `x` при цьому лишилася незмінною.

Виникає питання: що ж робити, якщо в тілі функції необхідно модифікувати глобальну змінну?

Модифікація глобальних змінних у тілі функції є небезпечним та може призвести до неочікуваних результатів під час виконання програми! Застосування такого підходу повинно бути обґрунтованим. У загальному випадку, для обміну даними між функцією і зовнішнім світом рекомендується використовувати механізм параметрів.

Для того, щоб у тілі функції явно вказати, які зі змінних є глобальними, використовують ключове слово **global**. Після цього слова вказують які змінні (через кому, якщо їх кілька) необхідно трактувати як глобальні.

```
x = 1          # x – глобальна змінна зі значенням 1

def func():
    global x # Вказуємо, що використовується глобальна x
    x = 222  # Змінимо значення x (глобальної)
    print(x) # Виводимо на екран глобальну змінну x

func()         # Викликаємо вищеописану функцію,
               # що виводить на екран x
print(x)       # Виводимо на екран глобальну змінну x
```

Результатом виконання цього коду буде

```
222
222
```

Функціональний тип

Опис функції призводить до створення об'єкту функціонального типу. Ім'я функції у цьому випадку є нічим іншим як змінною (посиланням на цей об'єкт), через яку можна здійснювати операцію виклику цієї функції.

Отже з точки зору Python не існує жодної концептуальної різниці між створенням функції і створенням звичайної змінної (об’єкту).

Як відомо зі змінними можна здійснювати операції, які визначаються типом літерала, до якого належить значення змінної. Наприклад, над числами можна проводити арифметичні операції (додавання, віднімання, множення і т.д.), над рядками і списками операції конкатенації, по-елементного проходження тощо. Єдина операція, яку можна здійснювати над функцією – викликати її.

Щоб краще зрозуміти вищезазначене, розглянемо приклад. Опишемо таку функцію

```
# Оголошуємо функціональний об'єкт (функцію) з іменем func
def func(p):
    print("Функція у точці {}".format(p))
```

Будемо вважати, що опис здійснено в окремому файлі, а приклад роботи з цією функцією здійснюється у інтерактивному режимі.

```
>>> func(5)       # Здійснимо виклик func для аргумента 5
Функція у точці 5
>>> func          # Виведемо на екран func
<function func at 0x01A484B0>
>>> func1 = func # func1 і func посилаються на один об'єкт
>>> func1         # Виведемо на екран func1
<function func at 0x01A484B0>
>>> func1(5)
Функція у точці 5
```

Отже, оскільки функції нічим не відрізняються від інших об’єктів Python, то функції можуть передаватися у якості аргумента іншим функціям. У тілі ж функції до такого параметру можна застосовувати операцію виклику

```
# Функція func має формальний параметр f який є функцією
# тобто посиланням на функціональний об'єкт
def func(..., f): # параметр f – фукнція
    ...
    x = f(...)     # Здійснюється виклик функції f
                   # що є аргументом функції func
    ...
```

Приклад 8.4. Для заданої функції потрібно побудувати таблицю її значень.

Розв'язок. Згідно з умовою задачі нам необхідно побудувати, наприклад, для функції $f = x^2$ для точок $\{0, 1, 2, 3, 4\}$ таку таблицю

x	0	1	2	3	4
$f = x^2$	0	1	4	9	16

Звичайно цю задачу можна реалізувати для конкретної функції. Проте зробимо універсальну функцію, що буде будувати таблицю для довільної заданої функції.

Отже вхідними параметрами у нашу функцію буде список точок і функція, для якої буде будуватися таблиця.

Таблицю оформимо у вигляді словника, у якій ключами будуть точки, а значеннями – значення функції у відповідній точці.

```
import math

def table(l, f):     # l – список точок, f – функція
    d = {}           # Словник d – таблиця значень функції
    for x in l:      # Проходимо по всіх x
        d[x] = f(x) # Додаємо пару (x, f(x)) у словник
    return d         # Повертаємо таблицю (словник)
```

```
# Створюємо список точок, що є розбиттям відрізка
# [0, pi] на 5 частин
l = [x * math.pi / 10 for x in range(6)]
# Будуємо таблицю значень функції sin для заданого списку
D = table(l, math.sin)

# Виводимо результат на екран у вигляді sin(0) = 0
for x in l:
    print("sin(%f) = %f" % (x, D[x]))
```

Результат виконання цієї програми

```
sin(0.000000) = 0.000000
sin(0.314159) = 0.309017
sin(0.628319) = 0.587785
sin(0.942478) = 0.809017
sin(1.256637) = 0.951057
sin(1.570796) = 1.000000
```

Як і звичайну змінну, функцію можна створювати у тілі іншої функцій. Більше того, створену функцію можна повернути у ролі результату за допомогою оператора **return** та використовувати отриману функцію у головній програмі.

Приклад 8.5. Обчислимо значення виразу

$$1 + x^2 + \cdots + x^n$$

для заданих дійсного числа x і натурального n.

Розв'язок. Опишемо функцію `pow_nat()`, що будує функцію, яка обчислює степінь порядку n для заданого аргументу x.

```
def pow_nat(n):

    def tmp_pow(x):
        res = 1
        for i in range(n):
            res *= x
        return res

    return tmp_pow

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

```
res = 0
for k in range(n + 1):
    res+= pow_nat(k)(x)
print(res)
```

Звернемо вашу увагу на те, що функція `pow_nat`, з одним аргументом повертає у якості результату функцію $f(x) = x^k$. Тобто інструкцію `pow_nat(k)` можна розглядати, як ім'я такої функції. Отже, щоб обчислити значення x^k, для заданого x використовуємо інструкцію `pow_nat(k)(x)`.

Анонімні функції

Анонімні функції (або лямбда-функції), це особливі функції, що описуються у місці їхнього виклику і не отримують імені (ідентифікатора) для доступу до них.

У Python анонімні функції можуть складатися лише з одного виразу.

Створюються анонімні функції з допомогою лямбда-виразу, що має такий синтаксис

```
lambda args: result
```

тут **`lambda`** – ключове слово, що вказує на опис анонімної функції, `args` – послідовність аргументів (параметрів) функції. `result` – результат, що повертає lambda-функція.

Як і під час створення звичайної функції, під час створення анонімної функції створюється об'єкт функціонального типу. Проте lambda-функції не отримують ідентифікатора для їхнього виклику. Основне призначення лямбда-функцій – використовуватися у ролі аргументу іншої функції.

Суттєвою перевагою лямбда-функцій над звичайними функціями є те, що вони виконуються значно швидше.

```
>>> (lambda x, y: x + y)(2, 3) # одночасне створення і виклик
5
>>> f = lambda x, y: x + y #f – посилання на лямбда-функцію
>>> f(2, 3)          # Виклик лямбда-функції через посилання f
5
```

Розглянемо застосування лямбда-функції до прикладу 8.4, у якому будувалася таблиця значень для заданої функції. Опис функції `table` опишемо схематично, оскільки він нічим не відрізняється від того, що наведений у прикладі 8.4**.** Побудуємо таблицю значень для функції $f = x^2$, для чого скористаємося відповідною лямбда-функцією. Отже програма буде мати вигляд

```
def table(l, f):
    ...

# l – розбиття відрізка [0, 1] на 10 частин
l = [x / 10.0 for x in range(11)]
# Будуємо таблицю значень функції x**2
D = table(l, lambda x: x**2)# у якості другого аргументу
                            # використовується лямбда-вираз
# Виводимо результат на екран у вигляді f(0.0) = 0.0
for x in l:
    print("f(%f) = %f" % (x, D[x]))
```

Рекурсія

Рекурсія – спосіб визначення об'єкту (або методу) попереднім описом одного чи кількох його базових випадків, з наступним описом на їхній основі загального правила побудови об'єкту.

З рекурсією ми часто зустрічаємося у оточуючому житті: рекурсивні зображення, структура рослин та кристалів, рекурсивні розповіді, вірші або жарти.

Рекурсивна функція – метод визначення функції, при якому функція прямо або неявно викликає сама себе.

Розглянемо вищенаведене означення на прикладі.

Як ми знаємо послідовність чисел Фібоначчі визначається рекурентним співвідношенням другого порядку

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

Отже, якщо покласти, що функція $F(n)$ визначає n-те число послідовності Фібоначчі, то з вищенаведеного рекурентного співвідношення отримаємо, що послідовність Фібоначчі може бути визначена рекурсивною функцією:

$$\begin{cases} F(0) = 1,\ F(1) = 1, \\ F(n) = F(n-1) + F(n-2),\ n \geq 2. \end{cases}$$

Початкові значення $F(0) = 1, F(1) = 1,$ дуже важливі при визначенні рекурсивної функції. Якби їх не було, то рекурсія б стала нескінченною! Тому, описуючи рекурсивну функцію завжди треба переконуватися, що для всіх допустимих значень аргументів виклик рекурсивної функції завершиться, тобто що рекурсія буде скінченною.

Можна звернути увагу, на те, що на відміну від обчислення елементів послідовності заданої рекурентним співвідношенням, де обчислення відбувається він тривіального (початкового) елементу до шуканого, рекурсивна функція починає обчислення від шуканого.

Кількість вкладених викликів функції називається **глибиною рекурсії**. Наприклад, глибина рекурсії при знаходженні $F(5)$ буде 4.

Опишемо стандартний алгоритм опису рекурсивної функції. Структурно рекурсивна функція на верхньому рівні завжди є розгалуженням, що складається з двох або більше альтернатив, з яких

- принаймні одна є термінальною (припинення рекурсії);
- принаймні одна є рекурсивною (тобто здійснює виклик себе з іншими аргументами).

```
def recursive_func(args):
    if terminal_case:        # Термінальна гілка
       ...
       return trivial_value # Тривіальне значення
    else:                    # Рекурсивна гілка
       ...
       # Виклик функції з іншими аргументами
       return expr(recursive_func(new_args))
```

Для прикладу опишемо рекурсивну підпрограму для обчислення чисел Фібоначчі. У програмі цю функцію будемо позначати `Fib(n)`.

```
def Fib(n):
    if n == 0 or n == 1:              # Термінальна гілка
       return 1                       # Тривіальне значення
    else:                             # Рекурсивна гілка
       return Fib(n - 1) + Fib(n - 2) # Рекурсивний виклик
# Виклик рекурсивної функції
n = int(input("n = "))
print("Fib(%d) = %d" % (n, Fib(n)))
```

Рекурсія використовується, коли можна виділити **самоподібність** задачі. Розглянемо інший класичний приклад, який використовується у навчальній літературі для пояснення рекурсії.

Приклад 8.6. Описати рекурсивну функцію для знаходження факторіала натурального числа.

Розв'язок. Очевидно, що функцію $F(n) = n!$ можна задати у такому рекурсивному вигляді

$$\begin{cases} F(0) = 1, \\ F(n) = nF(n-1), \ n \geq 1. \end{cases}$$

Тоді програма, разом з рекурсивною функцією (у програмі будемо позначати її `Fact(n)`) буде мати такий вигляд

```
def Fact(n):
    if n == 0 :                 # Термінальна гілка
       return 1                 # Тривіальне значення
    else:                       # Рекурсивна гілка
       return n * Fact(n - 1)   # Рекурсивний виклик
# Виклик рекурсивної функції
n = int(input("n = "))
print("%d! = %d" % (n, Fact(n)))
```

Приклад 8.7. Описати рекурсивну функцію перевірки рядка на симетричність.

Розв'язок. Спочатку опишемо термінальні випадки. Для цього проаналізуємо найпростіші рядки. Очевидно, що порожній рядок або рядок, що складається з одного символа (наприклад, `""`, `"a"`) є симетричним.

Тепер побудуємо рекурсивний виклик. Очевидно, що якщо перший і останній символи рядка є однаковими і рядок без них є симетричним, то вихідний рядок є симетричним.

```
# Рекурсивна функція перевірки рядка s на симетричність
def is_symetric(s):
    if len(s) <= 1:     # якщо рядок s порожній або
                        # складається з одного символа
        return True     # то він симетричний
    else:
        # cond1 - умова, що 1-й і останній символи однакові
        cond1 = s[0] == s[len(s)-1]
        # cond2 - умова, що рядок без першого і останнього
        # символа є симетричним - рекурсивний виклик
        cond2 = is_symetric(s[1 : len(s) - 1])
        # рядок симетричний, якщо обидві умови істині
        return cond1 and cond2
# Виклик рекурсивної функції для введеного з клавіатури рядка
S = input("S = ")
sym = is_symetric(S)

if sym:
    print("Заданий рядок є симетричним")
```

```
else:
    print("Заданий рядок не є симетричним")
```

Питання про використання рекурсії дуже суперечливе і неоднозначне. З одного боку, рекурсивна форма як правило значно простіша і наглядніша. З іншого боку, рекомендується уникати рекурсивних алгоритмів, що можуть приводити до занадто великої глибини рекурсії, особливо у випадках, коли такі алгоритми мають очевидну реалізацію у вигляді звичайного циклічного алгоритму. З огляду на це, вищенаведений рекурсивний алгоритм визначення факторіала є прикладом скоріше того, як не треба застосовувати рекурсію.

Теоретично будь-яку рекурсивну функцію можна замінити циклічним алгоритмом (можливо з застосуванням стеку).

Функції-генератори

Функція-генератор – спеціальний об'єкт Python, що (подібно до генератор-виразів) будує послідовність елементів деякої послідовності з по-елементним доступом до її членів.

Подібно до генератор-виразу, функція-генератор не будує всю послідовність одразу, а повертає її члени при кожному зверненні. Ця особливість дуже корисна при обробці великого масиву даних, оскільки не потрібно завантажувати весь масив, що економить час та оперативну пам'ять.

Опис генератор-функцій має такий же синтаксис, як і звичайна функція, за виключенням того, що для повернення результату (тобто поточного члена послідовності) замість оператора

```
return result
```

використовується оператор

```
yield result
```

Оператор **`yield`** (на відміну від **`return`**) не лише повертає деяке значення, але й запам'ятовує стан функції (місце завершення, значення всіх локальних змінних). Під час наступного виклику генератор-функції її виконання починається з наступного після **`yield`** оператора. Таким чином, виконання програми перемикається від програми до генератор-функції і назад.

Приклад 8.8. Опишемо генератор-функцію побудови чисел Фібоначчі. Використаємо її для виведення перших N чисел Фібоначчі.

Розв'язок. Опишемо спочатку генератор-функцію, що будує перші N чисел послідовності Фібоначчі та використаємо її.

```
def Fib(n):
    F1 = F2 = 1
    yield F2
    yield F1
    for i in range(3, n+1):
        F2, F1 = F1, F1 + F2
        yield F1

N = int(input("N = "))
# Створюємо об'єкт генератор-функцію
generator = Fib(N)

for a in generator:
    print(a, end=" ")
```

Оскільки генератор-функція не будує одразу всієї послідовності, то можна описувати генератор-функції для визначення як завгодно великих членів необмеженої послідовності. Перепишемо вищенаведений приклад побудови необмеженої послідовності за допомогою генератор-функції

```
def Fib():
    F1 = F2 = 1
    yield F2
    yield F1
    while True:
        F2, F1 = F1, F1 + F2
        yield F1

N = int(input("N = "))
i = 1
for a in Fib():  # Використання генератор-функції Fib
    print(a, end=" ")
    if i >= N:
        break
    i += 1
```

У цьому прикладі ми використовували генератор-функцію у циклі **for**, який відповідав за ітерацію елементів послідовності. Проте, генератор-функції можна використовувати і поза межами циклу по колекції. Для цього використовується функція `next`, яка повертає наступне значення генератор-функції:

```
next(generator)
```

тут `generator` – деякий об'єкт генератор-функції.

Змінимо вищенаведену програму з використанням функції `next`. Оскільки опис генератор-функції лишиться без змін, то у програмі опишемо його схематично

```
def Fib():
    ...

# створення об'єкту генератор-функції
generator  = Fib()

N = int(input("N = "))
for i in range(N):
    # Виводимо на екран наступне значення
    print(next(generator))
```

Для генератор-функцій, що будують обмежену послідовність (як у найпершому прикладі для послідовності Фібоначчі) функція `next` генерує виключення `StopIteration`, якщо відбувається спроба звернутися до неіснуючого елементу послідовності. Тому рекомендується оточувати виклик функції `next` блоком **`try…except`**.

```
def Fib(n):
    ...

# створення об'єкту генератор-функції
generator  = Fib(10)
while True:
    try:
        print(next(generator))
    except StopIteration:
        break
```

Декоратори для функцій

Означення

Як було вище зазначено, функції можна передавати у інші функції у ролі аргументів, а також функції можуть бути результатами роботи функцій. Цей факт використовується для створення спеціальних функцій, які називаються декораторами.

Декоратор – це функція, що дозволяє змінити поведінку функції (отриманої у якості аргументу) не змінюючи коду самої функції.

Як правило декоратори використовуються для того, щоб додати додаткові можливості функціям. Власне сенс застосування декоратора і полягає у тому, що вони можуть додавати однакову поведінку різним функціям. Наприклад, широко використовуються декоратори, що обчислюють час роботи функції, перевіряють коректність аргументів функцій, контролюють безпеку даних під час виконання функції тощо.

Щоб створити декоратор потрібно створити функцію, яка у якості аргументу буде отримувати функцію, модифікувати отриману функцію у тілі декоратора та повернути модифіковану функцію. Розглянемо такий приклад

```
def decorator(function):

    # Створюємо нову функцію
    def decorated_function():
        print("Код, що буде виконано до виклику функції")
        function()  # Виклик функції, що декорується
        print("Код, що буде виконано після виклику функції")

    # Повертаємо декоровану функцію
    return decorated_function
```

Виникає запитання: як скористатися створеним декоратором? Визначимо функцію

```
def seyHello():
    print("Усім вітання від функції seyHello!")
```

Задекоруємо її з допомогою декоратора `decorator`. Для цього необхідно змінній `seyHello` (що є посиланням на функцію у пам'яті) присвоїти результат виконання декоратора `decorator` над нею:

```
seyHello = decorator(seyHello)
```

Результатом виклику функції

```
seyHello()
```

буде

```
Код, що буде виконано до виклику функції
Усім вітання від функції seyHello!
Код, що буде виконано після виклику функції
```

Для спрощення розуміння коду, декорування функцій під час їхнього опису, можна здійснювати за допомогою оператора @. Наприклад, вищенаведені опис функції `seyHello` та її декорування

```
def seyHello():
    print("Усім вітання від функції seyHello!")
seyHello = decorator(seyHello)
```

можна замінити рівносильною їм синтаксичною конструкцією

```
@decorator
def seyHello():
    print("Усім вітання від функції seyHello!")
```

Загальний опис декораторів

У попередньому прикладі декоратор `decorator` міг застосовуватися лише до функцій, з порожнім списком аргументів. Загальний шаблон декоратора, що може застосовуватися до функцій з будь-якою кількістю як позиційних так і ключових аргументів, має такий вигляд:

```
def decorator(function):
    # Створюємо нову функцію
    def decorated_function(*args, **kwargs):
        # Код, що буде виконано до виклику функції
        ...
        res = function(*args, **kwargs)  # Виклик функції
        # Код, що буде виконано після виклику функції
        ...
        return res
    # Повертаємо декоровану функцію
    return decorated_function
```

Приклад 8.9. Опишемо декоратор, що вимірює час виконання функції. Застосуємо його для перевірки часу обчислення чисел Фібоначчі з використанням нерекурсивного і рекурсивного варіантів.

Розв'язок. Опишемо спочатку декоратор. Тут буде використовуватися функція `clock()` з бібліотеки `time`, що повертає поточний системний час.

Будемо зчитувати та запам'ятовувати час до початку виконання функції та після її виконання та виводити різницю на екран.

```
from time import clock # підключення функції clock
def benchmark(f):
    def _benchmark(*args, **kw):
        #вимірюємо час перед викликом функції
        current_time = clock()
        rez = f(*args, **kw)  #викликаємо f
        #обчилюємо різницю у часі
        dt = clock() - current_time
        print('Час виконання функції %1.5f сек' % dt)
        return rez
    return _benchmark
```

Опишемо функції `Fib1(n)` для нерекурсивного і `Fib2(n)` для рекурсивного варіантів обчислення `n`-го числа Фібоначчі, та задекоруємо їх за допомогою вищенаведеного декоратора `benchmark`. Оскільки декоратор діє на функцію, то для рекурсивного варіанту ми змушені використати допоміжну функцію `FibRecursive(n)`, яка буде реалізовувати рекурсію, а функція `Fib2` буде лише викликати функцію `FibRecursive(n)`.

```
@benchmark
def Fib1(n):
    F1 = F2 = 1
    for i in range(2, n + 1):
        F2, F1 = F1, F1 + F2
    return F1

def FibRecursive(n):
    if n == 0 or n == 1:
        return 1
    else:
        return FibRecursive(n-1) + FibRecursive(n-2)

@benchmark
def Fib2(n):
    return FibRecursive(n)
```

Виклик функцій нічим не відрізнятиметься від звичайного виклику недекорованих функцій:

```
N = 30
print(Fib1(N))
print(Fib2(N))
```

Проте, результатом роботи програми буде:

```
Час виконання функції 0.00001 сек
1346269
Час виконання функції 0.61396 сек
1346269
```

Вкладені декоратори

До функції може бути застосовано кілька декораторів одночасно. Їх вказують перед описом функції. Вказані декоратори застосовуються до функції послідовно, починаючи з найближчого до опису функції декоратора. Тому таке застосування декораторів називається їхнім вкладенням. Отже, якщо маємо код

```
@decorator1
@decorator2
def func():
    ...
```

то це означатиме, що до функції `func` спочатку застосовується декоратор `decorator2`, а вже потім до їхнього результату декоратор `decorator1`.

Задачі для самостійної роботи

8.1. Два простих числа називаються "близнюками", якщо вони відрізняються одне від одного на 2 (наприклад, числа 41 та 43). Скласти програму виведення на друк усіх пар "близнюків" із відрізка $[n,2n]$, де n – задане ціле число, яке більше за 2. У програмі описати функцію, що визначає чи є задане натуральне число простим.

8.2. Дано натуральне число n та послідовність натуральних чисел $a_1, a_2, \dots a_n$. Показати всі елементи послідовності, які є:

a) повними квадратами;

b) степенями п'ятірки;

c) простими числами.

Вказівка. Визначити відповідні функції для перевірки, чи є число повним квадратом, степенем п'ятірки, простим числом.

8.3. Дано натуральне число n. Для чисел від 1 до n визначити всі такі, які можна зобразити у вигляді суми двох повних квадратів. Описати функцію, яка