РОЗДІЛ 4. ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ

4.1. Функції

Зазвичай реалізація складних задач містить величезні фрагменти коду. Зручним способом організувати великий фрагмент коду в більш зручні фрагменти є створення функцій. Функції в Python є основою при написанні програм.

Іноді функцію порівнюють з "чорним ящиком", коли відомо, що на вході і що при цьому на виході, а нутрощі "чорного ящика" часто бувають приховані.

Існує велика кількість вбудованих функцій. Наприклад функція *abs()*, приймає на вхід один аргумент – об’єкт числового типу і повертає абсолютне значення для цього об’єкта.

```
>>> abs (-9)
9
```

Результат виклику функції можна присвоїти змінній, використовувати його в якості операндів математичних виразів, тобто складати більш складні вирази.

```
x = abs (-15)
y = x +5

print('y=', y)
```

Результатом запуску даного коду буде:

```
20
```

Крім складання складних математичних виразів Python дозволяє передавати результати виконання функцій в якості аргументів інших функцій без використання додаткових змінних:

```
print(abs (-15))
```

Отримаємо:

```
15
```

Спочатку визначається абсолютне значення цілого числа -15, а потім за допомогою функції *print()* виводиться на екран результат розрахунку.

Можна реалізовувати власні функції.

Функція – це іменований фрагмент коду, відокремлений від інших. Вона може приймає будь-яку кількість будь-яких вхідних параметрів і повертати будь-яку кількість будь-яких результатів.

З функцією можна зробити дві речі:

– визначити;

– викликати.

Щоб визначити функцію, використовується наступна конструкція:

def ІМ'Я_ФУНКЦІЇ (ВХІДНІ_ПАРАМЕТРИ):
ФУНКЦІЯ

Імена функцій підкоряються тим же правилам, що і імена змінних (вони повинні починатися з літери або _ і містити тільки букви, цифри або _).

Функція може не містити параметри але круглі дужки все рівно необхідно вказувати:

```
def do_nothing():
...     pass
```

Тіло функції відділяється пробілами. Використання виразу *pass* відображає, що функція нічого не робить.

Щоб викликати функцію вказується її ім’я та дужки з параметрами:

```
do_nothing()
```

Всі дії в програмі виконуються послідовно зверху вниз. Це означає, що перш ніж використовувати ідентифікатор в програмі, його необхідно попередньо оголосити, присвоївши йому значення. Тому визначення функції має бути розташоване перед викликом функції.

Визначимо і викличемо функцію, яка не має параметрів і виводить на екран одне слово:

```
def salute():
    print('Hi!')
salute()
```

Отримаємо:

```
Hi!
```

Коли викликається функція *salute()*, Python виконує код, розташований всередині її опису. У цьому випадку він виводить одне слово і повертає управління основній програмі.

Параметри функції

Функція може приймати параметри та повертати значення. Параметри функції – звичайні змінні, якими функція користується для внутрішніх розрахунків. Якщо параметрів декілька – вони перераховуються через кому [5 – 10, 13].

Формальні параметри – параметри, що вказуються при оголошенні функції.

Фактичні параметри (аргументи) – параметри, що передаються в функцію при її виклику.

```
def print_numbers(limit):
    for i  in range (limit):
        print(i)

n=int(input('Введіть кількість елементів: '))

print_numbers(n)
```

Результатом запуску даного коду буде:

```
Введіть кількість елементів: 5
0
1
2
3
4
```

Значення, які передаються в функцію при виклику, називаються ***аргументами***. Коли функція викликається з аргументами, їх значення копіюються у відповідні параметри всередині функції.

Існують функції які просто щось виконують, наприклад вбудована функція *print()* яка виводить на екран певні значення:

```
n=15
print(n)
```

Отримаємо:

```
15
```

А є функції які повертають розраховане значення, що може бути привласнене змінній, наприклад вбудована функція *input():*

```
a=input('Введіть слово: ')
```

Отримаємо:

```
Введіть слово: Hi!
```

Для того щоб переглянути результат роботи такої функції потрібно скористатися функцією *print()*

```
a=input('Введіть слово: ')
print(a)
```

Отримаємо:

```
Hi!
```

При необхідності повернути результат роботи функції в програму, з якої вона викликалася, для її подальшого оброблення застосовується команда ***return***. Вираз, що стоїть після *return* буде повертатися в якості результату виклику функції.

Без аргументів *return* використовується для виходу з функції (інакше вихід відбудеться при досягненні кінця функції).

В Python функції здатні повертати кілька значень одночасно.

```
def PrintRoots(a, b, c):
    D = b**2 - 4 * a * c
    import math
    x1 = (-b + math.sqrt(D)) / 2 * a
    x2 = (-b - math.sqrt(D)) / 2 * a
    return x1, x2
```

```
print (PrintRoots(1.0, 0, -1.0))
```

Результатом запуску даного коду буде:

```
(1.0, -1.0)
```

Крім того, результати виконання функції можна привласнювати відразу декільком змінним:

```
x1, x2 = PrintRoots(1.0, 0, -1.0)
print("x1 =", x1, "\nx2 =", x2)
```

Результатом запуску даного коду буде:

```
x1 = 1.0
x2 = -1.0
```

Всередині функції може міститися довільна кількість *return*. Однак спрацює лише один з них.

```
def traffic_light (color):
    if color == 'red':
        return "STOP!"
    elif color == "green":
        return "GO!"
    elif color == 'yellow':
        return "GET READY!"
    else:
        return "Broken traffic light!"
```

Викликавши функцію *traffic_light()*, передавши їй в якості аргументу рядок 'blue'.

```
result = traffic_light('blue')
print(result)
```

Функція зробить наступне:
- присвоїть значення 'blue' параметру функції color;
- пройде по логічному ланцюжку if-elif-else;
- поверне рядок;
- присвоїть рядок змінній result.

Результатом буде:

```
'Broken traffic light!'
```

Функція може приймати будь-яку кількість аргументів (включаючи нуль) будь-якого типу. Вона може повертати будь-яку кількість результатів (також включаючи нуль) будь-якого типу. Якщо функція не викликає *return* явно, буде отримано результат *None*.

```
def do_nothing():
    pass
```

Результатом буде:

```
None
```

None – це спеціальне значення в Python, яке заповнює собою порожнє місце, якщо функція нічого не повертає. Воно не є булевим значенням *False*, незважаючи на те що схоже на нього під час перевірки булевої змінної.

Простори імен та області видимості

Кожна функція визначає власний простір імен. Якщо визначити змінну, яка називається *x* в основній програмі та іншу змінну *x* в окремій функції, то вони будуть посилатися на різні значення. В основній програмі визначається глобальний простір імен, а змінні що тут знаходяться називаються ***глобальними*** тобто до неї можна звернутися з будь якого місця програми, в тому числі і всередині функції. Змінна є ***локальною*** (видно тільки всередині функції), якщо значення їй присвоюється всередині функцій [5, 8, 9].

```
a = 3                                   # Глобальна змінна
y = 8                                   # Глобальна змінна

def func ():
    print ('func: глобальна змінна a = ', a)
    y = 5                               # Локальна змінна
    print ('func: локальна змінна y = ', y)

func ()                                 # Виклик функції func()

print ('??? y = ', y)                   # Відобразиться глобальна
                                        змінна
```

```
print ('глобальна змінна a = ', a)
print ('глобальна змінна y = ', y)
```

Результатом запуску даного коду буде:

```
func: глобальна змінна a =  3
func: локальна змінна y =  5
??? y =  8
глобальна змінна a =  3
глобальна змінна y =  8
```

Всередині функції можна звернутися до глобальної змінної *a* і вивести її значення на екран. Далі всередині функції створюється локальна змінна *y*, причому її ім’я збігається з ім’ям глобальної змінної – в цьому випадку при зверненні до *y* виводиться вміст локальної змінної, а глобальна залишається незмінною.

Щоб змінити значення глобальної змінної всередині функції використовується ключове слово *global*.

```
x = 50                                  # Глобальна змінна

def func():
    global x                            # Вказуємо, що x-глобальна
змінна
    print('x =', x)
    x = 2                               # Змінюємо глобальну змінну
    print('Замінюємо глобальне значення x на', x)
func()

x = 50
print('Значення x =', x)
```

Результатом запуску даного коду буде:

```
x = 50
Замінюємо глобальне значення x на 2
Значення x = 50
```

З академічної точки зору зміна глобальної змінної всередині функції порушує принципи модульності програми.

Імена функцій в Python є змінними, що містять адресу об’єкта типу функція, тому цю адресу можна привласнити іншій змінній і викликати функцію з іншим ім’ям.

```
def summa (x, y):
    return x + y

print(summa(5,6))

f = summa
v = f (10, 3)                # Викликаємо функцію з іншим ім'ям
print(v)
```

Результатом запуску даного коду буде:

```
11
13
```

Аргументи функцій

Функція може приймати довільну кількість аргументів або не приймати їх зовсім. В функцію можна передавати не лише окремі об’єкти але і колекції/послідовності (список, кортеж та ін.). крім того, аргументи можуть бути позиційними, іменованими, обов’язковими та не обов’язковими.

Позиційні аргументи

Найбільш поширений тип аргументів – це ***позиційні аргументи***, чиї значення копіюються у відповідні параметри згідно з порядком проходження.

```
def func(a, b, c):
   return a+b*c

print(func(1, 2, 3))                    # a = 1, b = 2, c = 3
```

Отримаємо:

```
7
```

Незважаючи на поширеність аргументів такого типу, у них є недолік, який полягає в тому, що потрібно запам’ятовувати значення кожної позиції.

Іменовані аргументи

Щоб уникнути плутанини з позиційними аргументами, можна вказати аргументи за допомогою імен відповідних параметрів. Порядок проходження аргументів в цьому випадку може бути іншим:

```
print (func(a =2, b = 1, c = 3))
```

Отримаємо:

```
5
```

Можна об’єднувати позиційні аргументи та іменовані аргументи.

```
print (func(2, 2, c = 3))
```

Отримаємо:

```
8
```

Якщо викликати функцію, що має як позиційні аргументи, так і іменовані аргументи, то позиційні аргументи необхідно вказувати першими.

Значення параметра за замовчуванням

Можна вказати значення за замовчуванням для параметрів. Значення за замовчуванням використовуються в тому випадку, якщо викликаючи функцію не було вказано відповідний аргумент.

```
def func(a, b, c=2):
    return a+b*c
```

Викликаючи функцію *func()* можна не передавати їй аргумент *c*:

```
print(func(1, 2))
```

Отримаємо:

```
5
```

Але якщо надати аргумент, він буде використаний замість аргументу за замовчуванням:

```
print(func(1, 2, 3))
```

Отримаємо:

```
7
```

Отримання позиційних аргументів

Якщо перед параметром у визначенні функції вказати символ *, то функції можна буде передати будь-яку кількість параметрів. Всі передані параметри зберігаються в кортежі.

У наступному прикладі args є кортежем параметрів, який був створений з аргументів, переданих у функцію *print_args()*:

```
def print_args(*args):
    # функція приймає будь-яку кількість параметрів

    print('Кортеж позиційних аргументів:', args)
```

Якщо викликати функцію без аргументів, то буде отримано порожній кортеж:

```
print_args()
Кортеж позиційних аргументів: ()
```

Всі аргументи, які будуть передані, виведуться на екран як кортеж *args*:

```
print_args(1, 2, 3, 'Hi!')
Кортеж позиційних аргументів: (1, 2, 3, 'Hi!')
```

Це корисно при написанні функцій на зразок *print()*, які приймають будь-яку кількість аргументів. Якщо у функції є також обов’язкові позиційні аргументи, **args* відправиться в кінець списку і отримає всі інші аргументи:

```
def print_more(numb_1, numb_2, *args):
    print(numb_1)
    print(numb_2)
    print(args)

print_more(1, 2, 3, 4, 5, 6)
```

Результатом запуску даного коду буде:

```
1
2
(3, 4, 5, 6)
```

При використанні * не потрібно обов’язково називати кортеж параметрів *args*, однак це поширена ідіома в Python.

Отримання іменованих аргументів

Можна використовувати **, щоб згрупувати іменовані аргументи в словник, де імена аргументів стануть ключами, а їх значення – відповідними значеннями в словнику. У наступному прикладі визначається функція *print_kwargs ()*, в якій виводяться її іменовані аргументи:

```
def print_kwargs(**kwargs):
    print('Іменовані аргументи:', kwargs)
```

Викликавши її, передавши кілька аргументів:

```
print_kwargs(a = 1, b = 2, c = 3)
```

Отримаємо наступний результат:

```
Іменовані аргументи: {'b': 2, 'c': 3, 'a': 1}
```

Усередині функції *kwargs* є словником.

Якщо використано позиційні аргументи та іменовані аргументи (**args* і ***kwargs*), вони повинні слідувати в цьому ж порядку. Як і у випадку з *args*, не обов’язково називати цей словник *kwargs*.

Документаційні рядки

Можна додавати документацію до власних функцій, модулів, класів, заключивши рядок на початку тіла функції у лапки. Вона називається ***рядком документації*** або ***документаційним рядком (docstring)***:

```
def func(anything):
    'Функція повертає введений аргумент'
    return anything
```

Як правило документація містить розгорнуту інформацію про те, що дана функція (модуль) виконує, які аргументи приймає та що повертає в результаті виконання, опис всіх констант (функцій, для модуля). Тож документація може бути досить великого розміру і щоб використати до такої інформації форматування та вивести багато рядків коментарів необхідно заключити документаційний рядок в три пари подвійних лапок.

```
def print_if_true(thing, check):
    '''
    Prints the first argument if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
    '''
    if check:
        print(thing)
print(help(print_if_true))
```

На відміну від звичайних коментарів, до документаційних рядків можна звернутися під час виконання програми. Для того щоб вивести рядок документації деякої функції, необхідно викликати функцію *help()*, передати їй ім’я функції, щоб отримати список всіх аргументів і відформатований рядок документації:

```
Help on function print_if_true in module __main__:

print_if_true(thing, check)
    Prints the first argument if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.

None
```

Щоб відобразити рядок документації без форматування:

```
def print_if_true(thing, check):
    '''
```

```
    Prints the first argument if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
    '''
    if check:
        print(thing)

print(func.__doc__)
```

Отримаємо:

```
    Prints the first argument if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
```

Рядок *__doc__* є внутрішнім ім'ям рядка документації як змінної всередині функції.

Потік виконання

З появою функцій програми перестали бути лінійними, в зв'язку з цим виникло поняття ***потоку виконання*** – послідовності виконання інструкцій, що складають програму.

Виконання програми, написаної на Python завжди починається з першого виразу, а наступні вирази виконуються один за іншим зверху вниз. Причому, визначення функцій ніяк не впливають на потік виконання, тому що тіло будь-якої функції не виконується до тих пір, поки не буде викликана відповідна функція.

Коли інтерпретатор, розбираючи вихідний код, доходить до виклику функції, він, обчисливши значення аргументів, починає виконувати тіло функції, що викликається і тільки після її завершення переходить до розбору наступної інструкції.

З тіла будь-якої функції може бути викликана інша функція, яка теж може в своєму тілі містити виклики функцій і т.д. Проте, інтерпретатор Python пам'ятає звідки була викликана кожна функція, і рано чи пізно, якщо під час виконання не виникне

ніяких винятків, він повернеться до вихідного виклику, щоб перейти до наступної інструкції.

```
def func1(name):
    print('Привіт, '+name)

def func2():
    return input('Введіть ім\'я ')

func1(func2())
```

Результатом запуску даного коду буде:

```
Введіть ім'я Alex
Привіт, Alex
```

Внутрішні функції

Можна визначити функцію всередині іншої функції:

```
def outer(a, b):
    def inner(c, d):
        return c + d
    return inner(a, b)

print(outer(4, 7))
```

Результатом запуску даного коду буде:

```
11
```

Внутрішні функції можуть бути корисні при виконанні деяких складних завдань більш ніж один раз всередині іншої функції. Це дозволить уникнути використання циклів або дублювання коду.

Анонімні функції: функція lambda()

В Python лямбда-функція – це анонімна функція, виражена одним виразом. Її можна використовувати замість звичайної маленької функції.

Для того щоб проілюструвати анонімні функції, спочатку створимо приклад, в якому використовуються звичайні функції. Визначимо функцію *edit_story()*. Вона має такі аргументи:

– words – список слів;

– func – функція, яка повинна бути застосована до кожного слова в списку words.

– stairs – список слів

– edit_story – функція, яку потрібно застосувати кожного слова списку (записує з великої літери кожне слово і додає знак оклику):

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

def enliven(word):
    return word.capitalize() + '!'

stairs = ['hi', 'hello', 'привіт']

(edit_story(stairs, enliven)
```

Результатом запуску даного коду буде:

```
Hi!
Hello!
Привіт!
```

Функцію *enliven()* можна замінити лямбда функцією:

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

stairs = ['hi', 'hello', 'привіт']

edit_story(stairs, lambda word: word.capitalize() + '!')
```

Результатом запуску даного коду буде:

```
Hi!
Hello!
Привіт!
```

Лямбда приймає один аргумент, який в цьому прикладі названий word. Все, що знаходиться між двокрапкою та закриваючою дужкою, є визначенням функції.

Часто використання справжніх функцій на зразок *enliven()* набагато прозоріше, ніж використання лямбда. Лямбда найбільш корисні у випадках, коли потрібно визначити багато дрібних функцій і запам'ятати усі їх імена.

4.2. Рекурсія

В мові програмування Python функція може викликати будь-яку кількість інших функцій. Функції також можуть викликати самі себе, тобто мають властивість рекурсивності.

Рекурсія – спосіб опису об'єктів або обчислювальних процесів через самих себе. Рекурсивне програмування дозволяє описати процес що повторюється без явного використання операторів циклу.

Багато математичних функцій можна описати рекурсивно. Класичним прикладом програмування рекурсії є задача знаходження $n!$.

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1)! & n > 0 \end{cases}$$

```
def factorial (n):
    if  n>0:
        return n* factorial(n-1)
    else:
        return 1

print(factorial (5))
```

Отримаємо:

```
120
```

$$x^n = \begin{cases} 1 & n = 0 \\ x * x^{n-1} & n > 0 \end{cases}$$

```
x=10
def rec_func (n):
    if  n>0:
```

```
        return x* rec_func (n-1)
    else:
        return 1

print(rec_func (5))
```

Отримаємо:

```
100000
```

Рекурсивна функція обов'язково повинна містити хоча б одну альтернативу, що не використовує рекурсивний виклик, тобто явне визначення для деяких значень аргументів функції, тобто умову виходу (закінчення рекурсивності), щоб не спричинити зациклення програми. Кожний (новий) виклик вимагає додаткової пам'яті з ресурсу програмного стека. Якщо кількість викликів (глибина рекурсії) надмірно велика, виникає переповнення сегмента стека і операційна система вже не може створити наступний примірник локальних об'єктів функції, що як правило, веде до аварійного завершення програми.

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{i}$$

```
def rec_func_2(n, i):
    if  i==n:
        return 1/n
    else:
        return 1/i+rec_func_2(n, i+1)

print(rec_func_2(5, 1))
```

Отримаємо:

```
2.283333333333333
```

Можна простежити, як працює функція *rec_func_2*, наприклад, для $n = 5$.

rec_func_2(5, 1);

При виконанні тіла функції сформується наступне:

$$\frac{1}{1} + \text{rec\_func\_2}(5,2)$$

Що знову змушує звернутися до функції rec_func_2(5, 2), що призводить до появи нового значення:

$$\frac{1}{2} + \text{rec\_func\_2}(5,3)$$

Після виконання ще двох звернень ситуація виявиться наступною:

$$\frac{1}{3} + \text{rec\_func\_2}(5,4)$$

$$\frac{1}{4} + \text{rec\_func\_2}(5,5)$$

Потім при черговому виклику функції rec_func_2(5, 5) рекурсивні звернення припиняться і буде повернено значення $\frac{1}{5}$. В результаті сформується така послідовність:

Значення $\frac{1}{5}$ буде передано до $\frac{1}{4} + \text{rec\_func\_2}(5,5)$ замість $\text{rec\_func\_2}(5,5)$, потім $\frac{1}{4} + \frac{1}{5}$ до $\frac{1}{3} + \text{rec\_func\_2}(5,4)$ замість $\text{rec\_func\_2}(5,4)$ і т.д.

В результаті отримаємо ряд:

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5}$$

Ця послідовність операторів і дає результат обчислення суми:

Сума =2.28333333

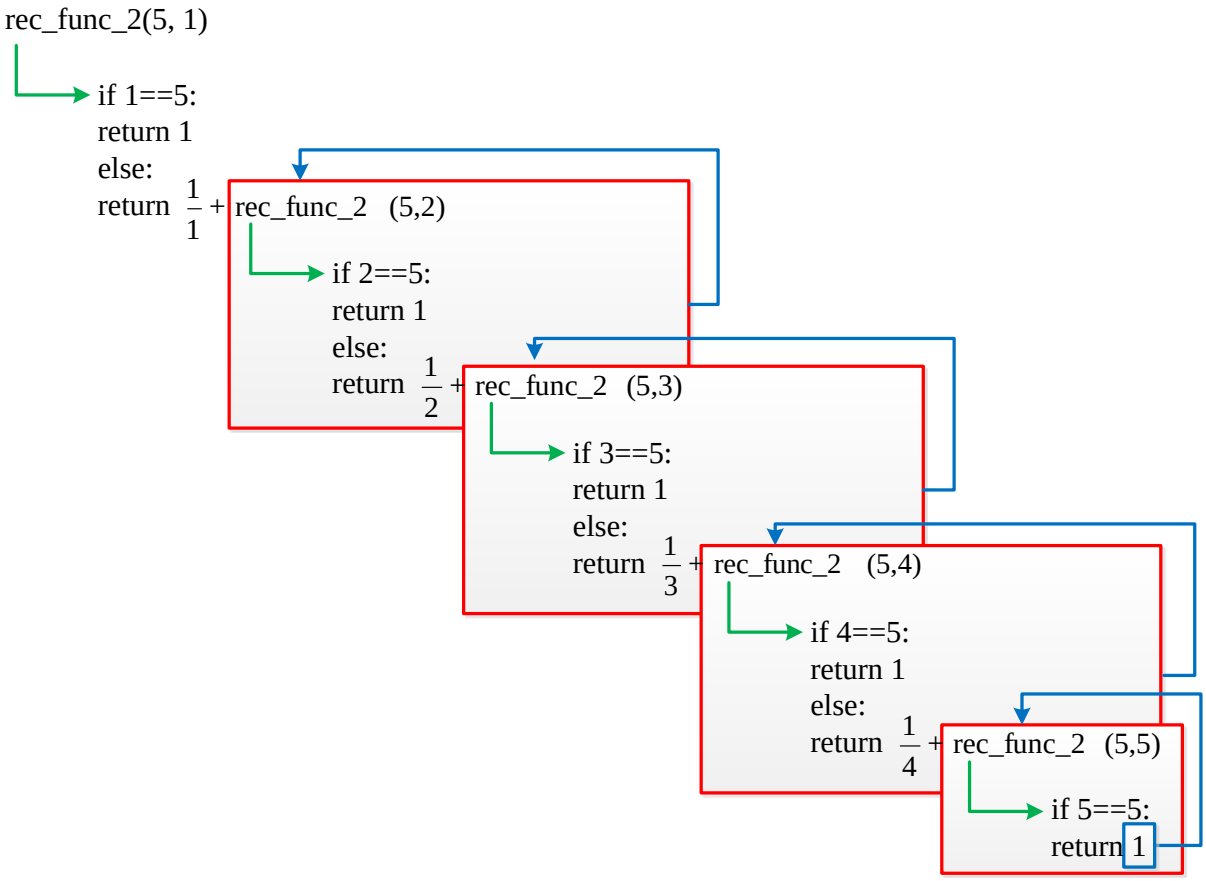

Рис.4.1. Графічне зображення роботи рекурсії

Ітерація і рекурсія засновані на керуючих структурах: ітерація використовує структуру повторення, рекурсія використовує структуру розгалуження.

```
def non_rec_func(n):
    S=0
    for i in range(1, n+1):
        S+=1/i
    return S

print(non_rec_func(5))
```

І ітерація, і рекурсія передбачають повторення: ітерація використовує структуру повторення явно, рекурсія – за допомогою повторних викликів функції.

Ітерація і рекурсія включають перевірку на завершення: ітерація завершується, коли перестає виконуватися умова продовження циклу, рекурсія завершується, коли розпізнається нерекурсивний випадок.

Як ітерація, так і рекурсія наближаються до завершення поступово.

Ітерація з її перевіркою повторення продовжує виконувати тіло циклу, поки умова продовження циклу не буде порушено. Рекурсія продовжує виробляти більш прості варіанти початкової задачі, поки не буде досягнутий нерекурсивний випадок.

І ітерація, і рекурсія може відбуватися нескінченно: ітерація потрапляє в нескінченний цикл, якщо умова продовження циклу ніколи не стає хибною; рекурсія триває нескінченно, якщо крок рекурсії не редукує задачу таким чином, що задача сходиться до нерекурсивного випадку.

Рекурсія має негативні сторони. Вона багато разів ініціалізує механізм виклику функції і збільшує пов’язані з ним витрати процесорного часу і пам’яті (кожне рекурсивне звернення створює копію її параметрів і локальних об’єктів). Ітерація зазвичай відбувається в межах функції, так що тут немає витрат на повторні виклики функції і додаткове виділення пам’яті. Налагодження рекурсивної функції викликає великі труднощі, ніж налагодження ітераційної функції.

Будь-яка проблема, яка може бути вирішена рекурсивно, може бути також вирішена і ітераційно (не рекурсивно).

Рекурсивний підхід краще ітераційного в тих випадках, коли рекурсія природніше відображає математичну сторону задачі і призводить до програми, яка простіше для розуміння.

Іншою причиною для вибору рекурсивного рішення є те, що ітераційне рішення може не бути очевидним.

4.3. Модульність в Python

Якщо код програми є складним, розбиття її на окремі функції допомагає спростити його для візуального сприйняття. Якщо цього недостатньо, є сенс винести частину функцій та пов’язаних з ними оголошень за межі основного файлу програми.

Такі додаткові файли з кодом, що використовується в програмі, називаються модулями. Найчастіше вони містять оголошення функцій та констант, які далі можуть бути підключені (імпортовані) в головну програму і вільно в ній використовуватися. Об’єкти з модуля можуть бути імпортовані в

інші модулі. Файл утворюється шляхом додавання до імені модуля розширення *.py*. При імпорті модуля інтерпретатор шукає файл спочатку в поточному каталозі, потім в каталогах, зазначених у змінній оточення *PYTHONPATH*, потім в залежних від платформи шляхах за замовчуванням, а також в спеціальних файлах з розширенням *'.pth'*, які лежать в стандартних каталогах. Можна внести зміни в *PYTHONPATH* і в *'.pth'*, додавши туди свій шлях. Каталоги, в яких здійснюється пошук, можна подивитися в змінній *sys.path*.

Великі програми, як правило, складаються з стартового файлу – файлу верхнього рівня, і набору файлів-модулів. Головний файл займається контролем програми. У той же час модуль – це не тільки фізичний файл. Модуль являє собою колекцію компонентів. У цьому сенсі модуль – це простір імен, – *namespace*, і всі імена всередині модуля ще називаються атрибутами – такими, наприклад, як функції і змінні [5, 8, 9, 13].

Є велика кількість вбудованих модулів, що дозволяють виконувати складні математичні операції (math), працювати з датами (datatime)/часом (time), випадковими числами (random), операційною та файловою системами (os) та ін.

Модуль math. Математичні функції

Модуль *math* надає додаткові функції для роботи з числами, а також стандартні константи. Перш ніж використовувати модуль, необхідно підключити його за допомогою інструкції:

import math

Модуль *math* надає наступні стандартні константи:
pi – повертає число n.
e – повертає значення константи *e*.

```
import math

print(math.pi)
print(math.e)
```

Отримаємо:

```
3.141592653589793
```

```
2.718281828459045
```

Основні функції для роботи з числами:

sin (), ***cos ()***, ***tan ()*** – стандартні тригонометричні функції (синус, косинус, тангенс). Значення вказується в радіанах;

asin (), ***acos ()***, ***atan ()*** – зворотні тригонометричні функції (арксинус, арккосинус, арктангенс). Значення повертається в радіанах;

degrees() – перетворює радіани в градуси:

```
import math

print(math.degrees(math.pi))
```

Отримаємо:

```
180.0
```

radians() – перетворює градуси в радіани:

```
import math

print(math.radians(180.0))
```

Отримаємо:

```
3.1415926535897931
```

exp() – експонента;
log() – логарифм;
sqrt() – квадратний корінь:

```
import math

print(math.sqrt(100), math.sqrt(25))
```

Отримаємо:

```
10.0, 5.0
```

ceil() – значення, округлене до найближчого більшого цілого:

```
import math

print(math.ceil(5.49), math.ceil(5.50), math.ceil(5.51))
```

Отримаємо:

```
6.0, 6.0, 6.0
```

floor() – значення, округлене до найближчого меншого цілого:

```
import math

print(math.floor(5.49), math.floor(5.50), math.floor(5.51))
```

Отримаємо:

```
5.0, 5.0, 5.0
```

pow(Число, Степень) – підносить Число до Степені:

```
import math

print(math.pow(10, 2), 10 ** 2, math.pow(3, 3), 3 ** 3)
```

Отримаємо:

```
100.0, 100, 27.0, 27
```

fabs() – абсолютне значення:

```
import math

print(math.fabs(10), math.fabs(-10), math.fabs(-12.5))
```

Отримаємо:

```
10.0, 10.0, 12.5
```

fmod() – остача від ділення:

```
import math

print(math.fmod(10, 5), 10 % 5, math.fmod(10, 3), 10 % 3)
```

Отримаємо:

```
0.0, 0, 1.0, 1
```

factorial() – факторіал числа:

```
import math
print(math.factorial(5), math.factorial(6))
```

Отримаємо:

```
120, 720
```

Модуль random. Випадкові числа

Більшість програм роблять одне і те ж при кожному виконанні, тому говорять, що такі програми визначені. Визначеність хороша річ до тих пір, поки ми вважаємо, що одні й ті ж обчислення повинні давати один і той же результат. Проте, в деяких програмах від комп'ютера потрібно непередбачуваність. Типовим прикладом є ігри, але є маса інших застосувань: зокрема, моделювання фізичних процесів або статистичні експерименти.

Змусити програму бути дійсно непередбачуваною завдання не таке просте, але є способи змусити її здаватися непередбачуваною. Одним з таких способів є генерування випадкових чисел і використання їх у програмі.

У Python є вбудований модуль, який дозволяє генерувати псевдовипадкові числа. З математичної точки зору, вони не істинно випадкові.

Модуль random дозволяє генерувати випадкові числа. Перш ніж використовувати модуль, необхідно підключити його за допомогою інструкції:

import random

Основні функції:

random() – повертає псевдовипадкове дійсне число від 0.0 до 1.0:

```
import random

print(random.random())
print(random.random())
print(random.random())
```

Отримаємо:

```
0.42888905467511462
0.57809130113447038
0.20609823213950174
```

Числа, що видаються функцією *random()*, розподілені рівномірно – це означає, що всі значення рівноймовірні.

uniform(start, end) – повертає псевдовипадкове дійсне число в діапазоні від ***start*** до ***end***:

```
import random

print(random.uniform(0, 10))
print(random.uniform(0, 10))
```

Отримаємо:

```
1.6022955651881965
5.206693596399246
```

randint(start, end) – повертає псевдовипадкове ціле число в діапазоні від ***start*** до ***end***:

```
import random

print(random.randint(0, 10))
print(random.randint(0, 10))
```

Отримаємо:

```
10
6
```

randrange(start, end, step) – повертає випадковий елемент з числової послідовності. Параметри аналогічні параметрам функції *range()*. Саме зі списку, що повертається функцією *range()*, і вибирається випадковий елемент:

```
import random

print(random.randrange(10))
print(random.randrange(0, 10))
print(random.randrange(0, 10, 2))
```

Отримаємо:

```
9
1
8
```

choice(Послідовність) – повертає випадковий елемент з будь-якої послідовності (рядку, списку, кортежу):

```
import random

print(random.choice("string"))          # Випадковий символ з рядку
print(random.choice(["s", "t", "r"]))   # Випадковий елемент зі списку
print(random.choice(("s", "t", "r")))   # Випадковий елемент з кортежу
```

Отримаємо:

```
't'
's'
'r'
```

shuffle(Список, Число від 0.0 до 1.0) – перемішує елементи списку випадковим чином. Функція перемішує сам список і нічого не повертає. Якщо другий параметр не вказано, то використовується значення, яке повернене функцією *random()*.

```
import random

lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(lst)

print(lst)
```

Отримаємо:

```
[7, 1, 6, 10, 9, 4, 8, 3, 2, 5]
```

sample(Послідовність, Кількість елементів) – повертає список із зазначеної кількості елементів. У цей список потраплять елементи з послідовності, вибрані випадковим чином. Як послідовність – можна вказати будь-який об’єкт, що підтримує ітерації.

```
import random

print(random.sample("string", 2) )
```

```
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(random.sample(lst, 2))

print(lst)                                  # Сам список не змінюється

print(random.sample((1, 2, 3, 4, 5, 6, 7), 3) )
```

Отримаємо:

```
['s', 'g']
[8, 7]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 4, 3]
```

Імпорт з модулів та його види

Для того щоб отримати доступ до функцій або змінних/констант з модуля та використати їх в основній програмі – його необхідно підключити до програми. Це можна зробити за допомогою інструкції **import МОДУЛЬ**, де модуль це ім’я іншого файлу Python без розширення *.py*.

```
import math
```

Дана команда імпортує модуль *math*. Тепер необхідно викликати з нього одну з функцій. Для того, щоб звернутися до змінної або функції з імпортованого модуля необхідно вказати його ім'я, поставити крапку і вказати необхідне ім’я **МОДУЛЬ.ФУНКЦІЯ/КОНСТАНТА**.

```
import math

print (math.e)
```

Отримаємо:

```
2.718281828459045
```

Запис *math.e* означає, що значення *e* знаходиться в просторі імен модуля *math*.

```
import math

print(math.sqrt(9))
```

Отримаємо:

```
3.0
```

Дізнатися, які функції і константи визначені в модулі можна за допомогою функції ***dir()*** :

```
import math

print(dir(math))
```

Отримаємо:

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'trunc']
>>>
```

В результаті виконання цієї команди інтерпретатор вивів всі імена, визначені в цьому модулі. У їх числі є і змінна ***__doc__***, що дозволяє вивести опис модуля.

```
import math

print (math.__doc__)
```

Отримаємо:

```
This module is always available.  It provides access to the
mathematical functions defined by the C standard.
```

__doc__ є внутрішнім ім’ям рядка документації як змінної всередині функції.

```
import math

print (math.e.__doc__)
```

Отримаємо:

```
float(x) -> floating point number
```

```
Convert a string or number to a floating point number, if possible.
```

Крім того, дізнатися про функції, які містить модуль, можна через функцію ***help()***:

```
import math

print(help(math))
```

Отримаємо:

```
Help on built-in module math:
NAME
   math
DESCRIPTION
   This module is always available.  It provides access to the
   mathematical functions defined by the C standard.
FUNCTIONS
   acos(...)
      acos(x)

      Return the arc cosine (measured in radians) of x.
…
```

Якщо необхідно ознайомитися з описом конкретної функції модуля, то викликається довідка окремо для неї:

```
import math

print(help(math.sqrt))
```

Отримаємо:

```
Help on built-in function sqrt in module math:
sqrt(...)
   sqrt(x)

   Return the square root of x.
```

Імпорт окремої функції з модуля

В Python можна імпортувати окрему функцію з модуля за допомогою наступної конструкції:

from МОДУЛЬ import ФУНКЦІЯ/КОНСТАНТА

```
from math import sqrt

print(sqrt(9))
```

Отримаємо:

```
3.0
```

Таким чином, Python не створюватиме змінну *math*, а завантажить в пам’ять тільки функцію *sqrt()*. Тепер виклик функції можна робити, не звертаючись до імені модуля *math*.

Через кому можна перерахувати декілька функцій або змінних які необхідні.

```
from math import sqrt, factorial

print(sqrt(9))
print(factorial(9))
```

Отримаємо:

```
3.0
362880
```

Або вказати * і тоді можна звертатися до будь-яких функцій.

```
from math import *

print(sin(pi/2))
```

Отримаємо:

```
1.0
```

В якості параметра тригонометричні функції приймають значення кута в радіанах.

```
from math import *

print(help(sin))
```

Отримаємо:

```
Help on built-in function sin in module math:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).
```

Створення власних модулів

Щоб створити власний модуль необхідно зберегти файл з власним ім'ям **ІМ'Я.py** (для модулів обов'язково вказується розширення *.py*), що містить якийсь код (вміст модуля). Наприклад створимо модуль назвавши його *my_math*:

```
def my_func ():
    print('test')
```

```
import my_math                  # Імпорт модуля my_math

my_math. my_func()              # Виклик функції my_func
```

Результатом запуску даного коду буде:

```
test
```

Якщо необхідно імпортувати одноіменні функції з кількох модулів, для них можна задати псевдоніми. Тоді функція буде теж скопійована в поточний простір імен, але під іншою назвою за допомогою наступної конструкції:

from МОДУЛЬ import ФУНКЦІЯ/КОНСТАНТА as НОВЕ_ІМ'Я

Імпортування модуля виконує команди що містяться в ньому. Однак, повторне імпортування не приводить до виконання модуля, тобто він повторно не імпортується. Пояснюється це тим, що імпортування модулів в пам'ять – ресурсномісткий процес, тому зайвий раз Python його не виконує. Якщо було внесено зміни до модуля – необхідно його повторно імпортувати,

примусово вказати Python, що модуль вимагає повторного завантаження. Після виклику функції *reload()* із зазначенням в якості аргументу імені модуля, оновлений модуль завантажиться повторно.

```
import imp

imp.reload(my_math)
```

Результатом запуску даного коду буде:

```
test
<module 'mtest' from 'C:\\Python35-32\\mtest.py'>
```

Каталоги пошуку модулів

Для імпорту Python шукає файли, що зберігається в стандартному модулі *sys*, як змінну *path*. Можна отримати доступ до цього списку і змінити його (відрізняється для різних операційних систем).

```
import sys

for place in sys.path:    # path містить список шляхів пошуку модулів
    print(place)
```

Результатом запуску даного коду буде:

```
D:\Users\syad\AppData\Local\Programs\Python\Python35\Lib\idlelib
D:\Users\syad\AppData\Local\Programs\Python\Python35\python35.zip
D:\Users\syad\AppData\Local\Programs\Python\Python35\DLLs
D:\Users\syad\AppData\Local\Programs\Python\Python35\lib
D:\Users\syad\AppData\Local\Programs\Python\Python35
D:\Users\syad\AppData\Local\Programs\Python\Python35\lib\site-packages
```

Список *sys.path* містить шляхи пошуку, одержувані з наступних джерел:

– шлях до поточного каталогу з виконуваним файлом;

– значення змінної оточення PYTHONPATH. Для додавання змінної в меню Пуск необхідно обрати пункт Панель керування (або Налаштування | Панель управління). Обрати пункт Система. Перейти на вкладку Додатково і натиснути кнопку Змінні середовища. У розділі Змінні середовища користувача натиснути кнопку Створити. В поле Ім’я змінної ввести "PYTHONPATH", а в полі Значення змінної задати шлях до папок, модулів через крапку з комою.

Після цих змін перезавантажувати комп’ютер не потрібно, достатньо заново запустити програму;

– шляхи пошуку стандартних модулів;

– вміст файлів з розширенням *pth*, розташованих в каталогах пошуку стандартних модулів, наприклад, в каталозі D:\Users\syad\AppData\Local\Programs\Python\Python35\lib\site-packages. Назва файлу може бути довільною, головне, щоб розширення файлу було *pth*. Кожен шлях (абсолютний або відносний) повинен бути розташований на окремому рядку.

Каталоги повинні існувати, в іншому випадку вони не будуть додані в список *sys.path*.

При пошуку модуля список *sys.path* проглядається зліва направо. Пошук припиняється після першого знайденого модуля. Таким чином, якщо в каталогах D:\Users\\folder1 і D:\Users\folder2 існують однойменні модулі, то буде використовуватися модуль з папки D:\Users\folder1, оскільки він розташований першим у списку шляхів пошуку.

Список *sys.path* можна змінювати з програми за допомогою спискових методів. Наприклад, додати каталог в кінець списку можна за допомогою методу *append()*.

```
import sys

sys.path.append(r" D:\Users\folder1")          # Додаємо в кінець
списку
for place in sys.path:
    print(place)
```

Результатом запуску даного коду буде:

```
D:\Users\syad\AppData\Local\Programs\Python\Python35\Lib\idl
elib
D:\Users\syad\AppData\Local\Programs\Python\Python35\python
35.zip
D:\Users\syad\AppData\Local\Programs\Python\Python35\DLLs
D:\Users\syad\AppData\Local\Programs\Python\Python35\lib
D:\Users\syad\AppData\Local\Programs\Python\Python35
D:\Users\syad\AppData\Local\Programs\Python\Python35\lib\site
-packages
D:\Users\folder1
```

Додати каталог в початок списку – за допомогою методу *insert()*.

```
import sys

sys.path.insert(0, r" D:\Users\folder2")        # Додаємо на
початку списку
for place in sys.path:
    print(place)
```

Результатом запуску даного коду буде:

```
D:\Users\folder2

D:\Users\syad\AppData\Local\Programs\Python\Python35\Lib\idl
elib
D:\Users\syad\AppData\Local\Programs\Python\Python35\python
35.zip
D:\Users\syad\AppData\Local\Programs\Python\Python35\DLLs
D:\Users\syad\AppData\Local\Programs\Python\Python35\lib
D:\Users\syad\AppData\Local\Programs\Python\Python35
D:\Users\syad\AppData\Local\Programs\Python\Python35\lib\site
-packages
D:\Users\folder1
```

Символ *r* перед лапками дозволяє не інтерпретувати спеціальні послідовності. Якщо використовуються

звичайні рядки, то необхідно подвоїти кожен слеш в шляху:

```
sys.path.append ("D:\\Users\\folder1\\folder2\\folder3")
```

Пакети

Коли модулів стає забагато, виникає необхідність групувати їх далі. Для цього файли модулів розкладаються по папках.

Відомо, що інтерпретатор шукає модулі в поточній папці та у спеціально призначеному для цього місці, отже необхідно якимось чином показати йому, що папка поряд з вашою програмою – не просто папка з файлами, а містить модулі для підключення. Для цього в папці повинен знаходитися файл *__init__.py* – він може бути порожнім, але сама його наявність сигналізує інтерпретатору, що папка із ним є пакетом модулів і може використовуватися в програмі [5, 8, 9, 13].

Пакетом називається каталог з модулями, в якому розташований файл ініціалізації ***__init__.py***.

Як і модулі, пакети створюють нові простори імен:

```
import my_package.my_math
# Модуль my_math шукатиметься в пакеті my_package

print(my_package.my_math.exp(1))
```

Отримаємо:

```
2.718281828459045
```

Всі розглянуті види імпорту поширюється також на пакети. Лише в іменах додається додатковий елемент через крапку – назва пакету. Пакети можуть вкладатися в інші пакети, аналогічно додаючи нові простори імен.

Як і модулі, пакети можуть містити код, який буде виконано під час ініціалізації пакету, – він записується в самому файлі __init__.py

Завдання на комп'ютерний практикум

1. Числа m та k ($3 \le k \le 10$) вводяться з клавіатури. Згенерувати та вивести на екран m цілих (дійсних) випадкових чисел з проміжку, вказаному у пункті а. Виведення на екран здійснювати по k чисел у рядку.
2. Розробити програму, дотримуючись таких вимог:

– число n (кількість елементів списку) – іменована константа;

– елементи списку – псевдовипадкові числа, згенеровані на інтервалі [a, b], де a і b вводяться з клавіатури (a<b);

– усі вхідні дані і також елементи списку виводяться на екран.

1	В одновимірному масиві (списку), що складається з n дійсних елементів, обчислити: 1) суму від'ємних елементів; 2) добуток елементів списку, розташованих між максимальним і мінімальним елементами.

3. Розробити програму, дотримаючись таких вимог:

– розміри масиву n і m – ввести з клавіатури;

– елементи масиву – псевдовипадкові числа, згенеровані на інтервалі [a, b], де a і b (a<b) вводяться з клавіатури;

– усі вхідні та вихідні дані і також елементи початкової матриці та отриманої виводити на екран.

1	Реалізувати програму, яка міняє місцями перший і останній стовпці квадратної матриці.

Запитання для самоконтролю

1. Яким чином можна згенерувати випадкове число?
2. Для чого існує функція random()?
3. Яким чином генеруються цілі випадкові числа на певному інтервалі?
4. Як згенерувати дійсні випадкові числа на певному інтервалі?
5. Що називають функцією?
6. Як відбувається звернення до функції?
7. Чи кожна функція повинна мати оператор повернення?
8. Що таке локальні змінні?
9. Що таке глобальні змінні?

10. Що таке фактичні параметри функції?
11. Що таке формальні параметри?
12. Чи можуть ідентифікатори фактичних і формальних параметрів співпадати?
13. Чи обов’язково кількість фактичних і формальних параметрів повинні співпадати?
14. Чи може глобальна змінна бути розташована у тілі програми?
15. Чи можна у середині однієї функції оголошувати іншу функцію?
16. Що таке документаційні рядки?