

# ОСНОВИ ПРОГРАМУВАННЯ МОВОЮ Python

# Урок 4

## Функції

### ЗМІСТ

<b>1. Функції і модулі .....</b>	<b>4</b>
1.1. Що таке функція? Цілі та завдання функції .....	4
1.2. Вбудовані функції.....	7
1.3. Математичні функції та випадкові числа.....	9
1.4. Синтаксис оголошення функцій.....	14
1.5. Аргументи та значення, що повертаються.....	17
<b>2. Розширені прийоми в роботі з функціями .....</b>	<b>25</b>
2.1. Аргументи за замовчуванням, аргументи-ключі .....	28
2.2. Область видимості, правило LEGB.....	33
2.3. Локальні та глобальні змінні у функціях.....	36
2.4. Функції як об'єкти першого класу.....	43
2.5. Рекурсія.....	48

<b>3. Функціональне програмування.....</b>	<b>53</b>
3.1. Що таке функціональне програмування?.....	53
3.2. Анонімні функції lambda.....	58
3.3. Функції вищих порядків.....	63
3.4. Функції map(), filter(), zip().....	69
3.5. Модуль functools.....	78
3.6. Замикання .....	89
3.7. Каррінг .....	95
<b>4. Декоратори .....</b>	<b>99</b>

# 1. Функції і модулі

## 1.1. Що таке функція? Цілі та завдання функції

Раніше наші програми являли собою єдиний, функціонально неподільний блок коду, який послідовно оброблявся інтерпретатором відповідно до використаних умовних блоків, циклів і т. п. Однак у реальних завданнях, алгоритми яких часто досить складні та об'ємні, рішення розбивається на окремі кроки (кожен з яких часто має власний алгоритм рішення). При цьому в загальному алгоритмі ці кроки (логіка реалізації цієї задачі) можуть повторюватися.

Наприклад, у нас є дані про студента (припустимо у вигляді списку `[name, age, score]`).

```
student=['Bob', 19, 95]
```

Меню нашої програми пропонує користувачеві вивести інформацію про студента, змінити інформацію про студента. Перед тим, як виконати дії пункту меню, програма запитує логін користувача для перевірки прав доступу до цієї дії (чи є логін у списку користувачів).

```
users=['admin', 'teacher', 'manager']
template="Name: {}; age: {}; scores: {}."

while True:
    print("1-print info")
    print("2-mofify info")
    print("3-exit")
    userChoice=input("Select menu item: ")
```

```

if userChoice=="1":
    user=input("Login:")
    if user in users:
        print("Current information:")
        print(template.format(student[0],
                                student[1], student[2]))
elif userChoice=="2":
    if user in users:
        print("Current information:")
        print(template.format(student[0],
                                student[1], student[2]))
        userAnsw=input("Change this info:y-n?")
        if userAnsw=="y":
            name=input("Input new name:")
            age=int(input("Input new age:"))
            scores=input("Input new scores:")
            student[0]=name
            student[1]=age
            student[2]=scores
            print(template.format(student[0],
                                    student[1], student[2]))
elif userChoice=="3":
    print("Bye!")
    break
else:
    print("Error: wrong menu item!")

```

У такому випадку блок коду, який здійснює таку перевірку користувача та виведення поточної інформації про студента, повторюватиметься перед кожним блоком коду для реалізації пункту меню.

```

if user in users:
    print("Current information:")
    print(template.format(student[0],
                            student[1], student[2]))

```

А якщо пунктів меню не два, а набагато більше? Саме для того, щоб забезпечити повторення логіки в основній програмі без дублювання рядків коду, які реалізують цю логіку, в програмуванні передбачені функції. Функції є невід'ємною частиною більшості мов програмування.

Функція — це іменований блок програмного коду, який використовується для виконання деякого набору дій (відповідно до алгоритму) і організований таким чином, щоб багаторазово використовуватися.

Для запуску такого фрагмента коду у потрібному місці основної програми використовується лише ім'я функції (відбувається виклик функції, тобто запуск нашого, раніше написаного блоку коду тіла функції). Функції можна викликати у будь-якому місці програми Python, у тому числі викликати функції всередині інших функцій.

Використання функцій у програмуванні забезпечує такі переваги:

- функції дозволяють виконувати один і той самий фрагмент коду кілька разів без дублювання рядків коду;
- функції розбивають довгі програми на дрібніші компоненти, що підвищує читабельність коду і робить більш зрозумілою логіку алгоритму;
- функції можуть бути загальними та використовуватися іншими програмістами, які просто використовуватимуть їх як «чорну скриньку» для вирішення певного завдання, не замислюючись про особливості її реалізації.

Функції забезпечують кращу модульність (структурованість) для вашого додатку та ефективність у повторному використанні коду.

## 1.2. Вбудовані функції

В Python існує велика кількість вбудованих функцій. Вони перераховані нижче в алфавітному порядку.

<b>A</b> <code>abs()</code> <code>aiter()</code> <code>all()</code> <code>any()</code> <code>anext()</code> <code>ascii()</code>	<b>E</b> <code>enumerate()</code> <code>eval()</code> <code>exec()</code>	<b>L</b> <code>len()</code> <code>list()</code> <code>locals()</code>	<b>R</b> <code>range()</code> <code>repr()</code> <code>reversed()</code> <code>round()</code>
<b>B</b> <code>bin()</code> <code>bool()</code> <code>breakpoint()</code> <code>bytearray()</code> <code>bytes()</code>	<b>F</b> <code>filter()</code> <code>float()</code> <code>format()</code> <code>frozenset()</code>	<b>M</b> <code>map()</code> <code>max()</code> <code>memoryview()</code> <code>min()</code>	<b>S</b> <code>set()</code> <code>setattr()</code> <code>slice()</code> <code>sorted()</code> <code>staticmethod()</code> <code>str()</code> <code>sum()</code> <code>super()</code>
<b>C</b> <code>callable()</code> <code>chr()</code> <code>classmethod()</code> <code>compile()</code> <code>complex()</code>	<b>G</b> <code>getattr()</code> <code>globals()</code>	<b>N</b> <code>next()</code>	<b>T</b> <code>tuple()</code> <code>type()</code>
<b>D</b> <code>delattr()</code> <code>dict()</code> <code>dir()</code> <code>divmod()</code>	<b>H</b> <code>hasattr()</code> <code>hash()</code> <code>help()</code> <code>hex()</code>	<b>O</b> <code>object()</code> <code>oct()</code> <code>open()</code> <code>ord()</code>	<b>V</b> <code>vars()</code>
	<b>I</b> <code>id()</code> <code>input()</code> <code>int()</code> <code>isinstance()</code> <code>issubclass()</code>	<b>P</b> <code>pow()</code> <code>print()</code> <code>property()</code>	<b>Z</b> <code>zip()</code>
			<b>_</b>

Рисунок 1

З багатьма з них ми вже знайомі. Наприклад, вбудована функція `abs(x)` приймає на вхід число `x` і повертає його абсолютне значення. Вбудована функція `chr(i)` повертає рядок, що представляє символ, код якого є цілим числом

і. А для перетворення рядка на ціле число ми використували функцію `int(s)`.

```
x=int(input("Input Unicode code point: "))
if x<0:
    x=abs(x)

print(chr(x))
```

Як видно з прикладу, для використання вбудованих функцій ми просто вказуємо їх ім'я (тобто виконуємо виклик цієї вбудованої, потрібної нам функції) і, при необхідності, передаємо дані для їх роботи (у круглих дужках після імені).

Якщо програмісту необхідно дізнатися, як виглядає загальний синтаксис вбудованої функції і що вона повертає, то можна викликати вбудовану функцію `help()`, на вхід якої потрібно передати ім'я функції, яка нас цікавить.

Даний фрагмент коду можна написати окремо від основної програми, наприклад, в консолі або навіть в окремому файлі, щоб просто прочитати опис синтаксису і призначення потрібної функції.

Наприклад:

```
help(len)
```

```
Help on built-in function len in module builtins:
```

```
len(obj, /)
```

```
Return the number of items in a container.
```

Рисунок 2



### 1.3. Математичні функції та випадкові числа

Для вирішення математичних завдань у Python є вбудований модуль `math`, який містить набір методів та констант. Для роботи з вбудованими математичними функціями необхідно спочатку імпортувати модуль `math`.

```
import math
```

Після підключення до бібліотеки доступ до її функцій здійснюється таким чином:

```
math.ім'я_функції(...)
```

Розглянемо найпоширеніші функції модуля `math`. Однією з найбільш затребуваних функцій є функція округлення числа до найближчого цілого. Для цього завдання в модулі `math` передбачені дві функції:

- `ceil(x)` — повертає найближче ціле значення, яке більше або дорівнює числу `x` (округлення «вгору»);
- `floor(x)` — повертає найближче ціле значення, яке менше або дорівнює числу `x` (округлення «вниз»).

```
import math
x=0.23
y=1.87
z=3

print(math.ceil(x)) #1
print(math.ceil(y)) #2
print(math.ceil(z)) #3
print(math.floor(x)) #0
print(math.floor(y)) #1
print(math.floor(z)) #3
```

Якщо потрібно отримати цілу або дробову частину числа, то в модулі `math` також є відповідні функції: `modf(x)` — повертає дробову та цілу частину `float` числа `x` із збереженням знака числа `x`. Тип поверненого результату — також `float`.

```
x=-2.45
y=1.5
print(math.modf(x))  # (-0.4500000000000002, -2.0)
print(math.modf(y))  # (0.5, 1.0)
```

Для отримання цілої частини числа можна скористатися функцією `trunc(x)`:

```
x=-2.45
y=1.5
print(math.trunc(x))  #-2
print(math.trunc(y))  #1
```

Для отримання абсолютного значення (модуля) числа можна скористатися функцією `fabs(x)`. На відміну від вбудованої функції `abs()`, яка повертає модуль числа з таким самим типом, як і вихідне число, функція `fabs(x)` повертає результат типу `float`.

```
import math
x=-2
y=-2.5
z=4

print(math.fabs(x))  #2.0
print(math.fabs(y))  #2.5
print(math.fabs(z))  #4.0
```

```
print(abs(x)) #2
print(abs(y)) #2.5
print(abs(z)) #4
```

Для обчислення факторіалу цілого числа передбачено функцію `factorial(x)`. Якщо число `x` не ціле, то виникне виняток `ValueError`.

Для роботи із випадковими числами в Python використовується модуль `random`. Він включає безліч корисних функцій для генерації, роботи із випадковими числами та послідовностями (наприклад, випадкове перемішування елементів списку або вилучення випадкового елемента зі списку). На початку роботи необхідно підключити (імпортувати) модуль `random`, щоб надалі використовувати його функції.

```
import random
```

Розглянемо функції для генерації випадкових чисел та послідовностей випадкових чисел. Для генерації псевдовипадкових цілих чисел передбачені функції `randint()` і `randrange()`.

`randint(a, b)` — приймає лише два аргументи `a`, `b` — межі діапазону цілих чисел, з яких вибирається випадкове число. При цьому обидві межі включаються до діапазону, це означає, що випадкове число буде вибрано з відрізка `[a; b]`.

Числа можуть бути від'ємними, для цього межі діапазону мають бути від'ємними. Однак перше число завжди має бути менше другого (`a < b`).

```
import random
for i in range(3):
    print("step ", i+1)
    print(random.randint(1,5))
    print(random.randint(-3,2))
    print(random.randint(-10,-6))
```

```
step 1
4
0
-6
step 2
5
-2
-9
step 3
3
-3
-6
```

Рисунок 3

Функція `randrange()` може приймати від одного до трьох аргументів:

- якщо вказано один аргумент — `randrange(a)`, то результат — випадкове число від 0 до вказаного. Причому сам аргумент у діапазон генерації не входить: `[0; a)`;

**Примітка:** у записі `[0; a)` символ «`[`» означає, що значення 0 входить у проміжок (діапазон генерації), а символ «`)`» каже, що значення `a` в діапазон не входить, це означає, що генерація буде проводитись від 0 (включно) до `a-1`.

- за двох аргументів — `randrange(a, b)` працює аналогічно `randint()`, крім того, що верхня межа `b` не входить в діапазон генерації, тобто `[a; b)`;
- якщо вказано три аргумента — `randrange(a, b, c)`, то перші два є межами діапазону, а третій — крок генератора, тобто, наприклад, під час виклику `randrange(10, 18, 2)`, «випадково» число вибиратиметься з послідовності чисел `10, 12, 15, 17`.

```
import random

print(random.randrange(10))
print(random.randrange(-4, 3))
print(random.randrange(10, 20, 3))
```

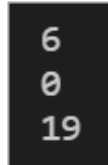


Рисунок 4

Для генерації випадкового цілого числа використовується функція `random()`. Ця функція викликається без аргументів, оскільки випадкове число генерується в діапазоні від `0` до `1`, але не включаючи `1`:

```
import random

for i in range(3):
    print("step ", i+1)
    print(random.random())
```

```

step 1
0.7424763426403829
step 2
0.9118237572852429
step 3
0.05515544230699532

```

Рисунок 5

Якщо ж нам необхідно отримати випадковий елемент з послідовності, наприклад, випадковий елемент списку, то можна скористатися функцією `choice(L)`, де `L` — це послідовність або її ім'я:

```

import random
L=[1,34,67,89]
for i in range(3):
    print(random.choice(L))
    print(random.choice(['red','blue','green','white']))

```

```

67
green
1
white
1
blue

```

Рисунок 6

## 1.4. Синтаксис оголошення функцій

Як і в інших мовах програмування, в Python можна створювати власні (користувацькі) функції. Робота з функціями користувача відбувається в два етапи: спочатку функцію потрібно створити (визначити, тобто написати

блок коду, що реалізує її логіку), а потім викликати в тому місці (або декількох місцях) коду основної програми (або в інших функціях), де це потрібно.

У Python функції визначаються за допомогою ключового слова `def`, після якого вказується ім'я функції та круглі дужки з вхідними параметрами (якщо вони є):

```
def ім'я_функції(вхідні_параметри):
    тіло_функції
```

Наприклад:

```
def printMsg():
    print("Hello!")
    print("Welcome!!!")
```

У нашому прикладі функція не приймає жодних вхідних даних (тому круглі дужки порожні, але вони обов'язкові) і не повертає жодних результатів, а просто виконує певну послідовність дій.

Ім'я функції має відповідати таким самим правилам, як і імена змінних, тобто починатися з літери або символу підкреслення «`_`», і містити лише літери, цифри та символ «`_`».

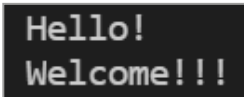
Як ми вже знаємо з прикладів використання вбудованих функцій, функції, за потреби, можуть отримувати дані (аргументи функції) і повертати результат. При цьому вхідні дані для функцій приходять не з клавіатури, файлу і т. п., а з основної (викликаючої їх) програми. Сюди ж вони повертають результат своєї роботи, якщо це передбачено цією функцією.

Рядки коду, що реалізують логіку роботи функції, називаються тілом функції та йдуть після знака «:» з відступом (як, наприклад, тіло циклу). Щоб викликати функцію, потрібно в коді основної програми вказати ім'я функції та дужки (з параметрами, якщо вони необхідні).

```
printMsg()
```

Оскільки в нашому прикладі у функцію ми не передаємо жодних вхідних даних (у нашої функції немає параметрів), дужки під час виклику також порожні:

```
def printMsg():  
    print("Hello!")  
    print("Welcome!!!")  
printMsg()
```



```
Hello!  
Welcome!!!
```

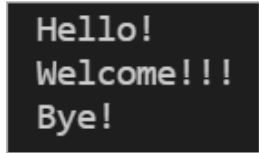
*Рисунок 7*

Коли функція викликається, потік виконання програми переходить до точки її визначення і починає виконувати тіло функції. Після того, як усі рядки коду з тіла функції виконані, потік виконання повертається в основну програму, у те місце, звідки функція викликала. Далі виконується наступний за викликом вираз (наступний рядок коду основної програми).

```
def printMsg():  
    print("Hello!")  
    print("Welcome!!!")
```



```
printMsg()
print("Bye!")
```



```
Hello!
Welcome!!!
Bye!
```

Рисунок 8

## 1.5. Аргументи та значення, що повертаються

При необхідності, і в залежності від особливостей задачі, що вирішується, функції можуть приймати дані для їх подальшої обробки всередині функції або для реалізації логіки алгоритму функції на основі їх значень.

Такі дані називаються параметрами функції. Параметри функції — це звичайні змінні, які функція використовує всередині свого коду (для внутрішніх дій згідно з деяким алгоритмом). Ім'я параметра вказується в круглих дужках під час оголошення функції:

```
def printMsg1(myMsg):
    print(myMsg)
```

Якщо параметрів у функції декілька, їх імена перераховуються в круглих дужках через кому.

```
def printMsg2(myMsg1, myMsg2):

    print(myMsg1)
    print(myMsg2)
```

Параметри представляють собою локальні змінні, яким надаються значення в момент виклику функції.

Конкретні значення, які передаються у функцію під час її виклику, називаються аргументами.

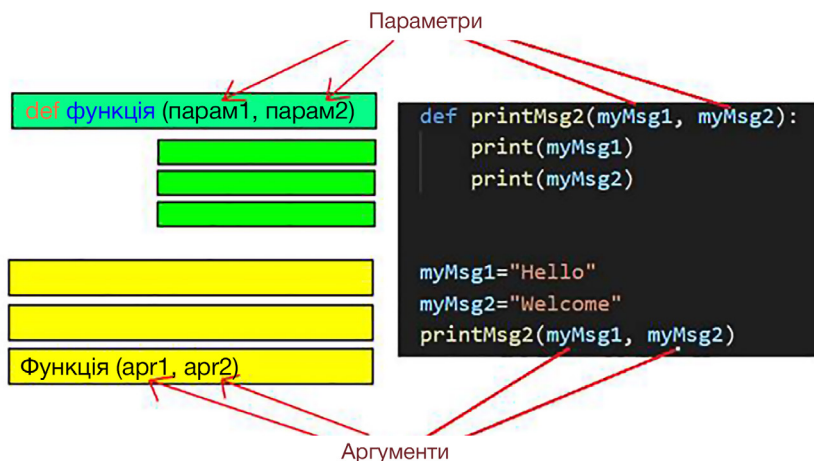


Рисунок 9

Коли функція викликається, то їй передаються аргументи. У нашому прикладі вказані змінні `myMsg1` і `myMsg2`. Проте фактично передаються не ці змінні, а їх значення, тобто рядки «Hello» та «Welcome». Тому аргументи також ще називають фактичними параметрами.

У нашому прикладі імена аргументів і параметрів збігаються, але це зовсім не обов'язково. Імена можуть бути різні, оскільки фактично в момент виклику функції програма копіює значення із змінних-аргументів у щойно створені змінні-параметри.

```
def calculate(a,b,c):
    print("Let's calculate!")
```

```
print (a+b*c**2)

x=5
y=2
z=3
calculate(x,y,z)
calculate(x,y,5)
calculate(1,7,10)
```

```
Let's calculate!
23
Let's calculate!
55
Let's calculate!
701
```

Рисунок 10

Як бачимо, аргументами можуть бути як імена змінних, так і літерали (значення).

Функції в наших прикладах просто виконували деякі дії над параметрами (виводили тексти повідомлень у консоль), не повертаючи жодного результату до основної програми. Проте часто є необхідність створювати функції, які не тільки виконують якісь дії, а й повертають обчислений всередині функції результат в основну програму.

У цьому випадку в основній програмі необхідно передбачити змінну, яка прийматиме результат роботи функції.

Наприклад, таким чином ми працювали із вбудованою функцією `input()`:

```
name=input("Input your name:")
```

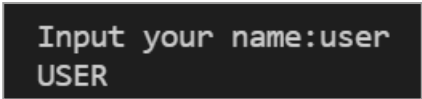
Таким же чином і функції користувача можуть передавати результат своєї роботи (повертати значення).

Вихід із функції та передача даних (деякого обчисленого всередині функції результату) відбувається у тому місці, звідки ця функція була викликана. Для виходу з функції у її тілі потрібно використовувати оператор **return**.

Якщо інтерпретатор, виконуючи тіло функції, зустрічає оператор **return**, він «забирає» значення змінної, ім'я якої зазначено після цієї команди, і «виходить» із функції відразу після цього (навіть у разі, якщо далі є ще команди).

Наприклад:

```
def modifyName(userName):  
    newName=userName.upper()  
    return newName  
  
name=input("Input your name:")  
print(modifyName(name))
```



```
Input your name:user  
USER
```

Рисунок 11

Якщо ми додамо рядки коду після команди **return newName** в тілі функції **modifyName(userName)**, то вони не будуть виконані, тому що програма «вийде» з функції після **return newName**:

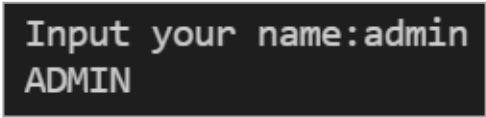
```
def modifyName(userName):  
    newName=userName.upper()
```

```

    return newName
    print("Hello!")

name=input("Input your name:")
print(modifyName(name))

```



```

Input your name:admin
ADMIN

```

Рисунок 12

Припустимо, що нам потрібно обчислити значення виразу, що обчислюється, за такою формулою:

Як бачимо, формула для обчислення результату різна і залежить від значення виразу. Тому, зручно створити окрему функцію, яка обчислює цей вираз, а потім перевіряти значення, яке поверне функція, в основній програмі та за результатами перевірки застосовувати одну чи другу формулу:

```

def calculate(a,b):
    return a+b

x=float(input("Input x:"))
z=float(input("Input z:"))

if calculate(x,z)<=0:
    y=5*x+2*z
else:
    y=10*x-3*z

print("y=",y)

```

```
Input x:2
Input z:4
y= 8.0
```

Рисунок 13

Припустимо, що у нас є список клієнтів і нам для заданого клієнта потрібно вивести його позиції у списку та підрахувати, скільки разів він зустрічається у списку, тобто, для частих клієнтів (тих, хто зустрічається у списку більше одного разу) діє деяка знижка.

Створимо функцію, яка виводить позиції елементів (клієнтів) у списку та підраховує (повертає) кількість входжень.

В основному коді використовуємо результат роботи цієї функції для визначення ситуації, чи буде знижка для цього клієнта:

```
def checkCustomer(customer, customers):
    result=0
    print("Client's positions in the list:")

    for i in range(len(customers)):
        if customers[i]==customer:
            print(i)
            result+=1
    return result

customerList=['Bob', 'Anna', 'Joe', 'Bob', 'Nick']

myCustomer='Bob'

if checkCustomer(myCustomer, customerList)>1:
    print("Customer", myCustomer, "will get a discount")
```

Отриманий результат для клієнта «Bob»:

```
Client's positions in the list:
0
3
Customer Bob will get a discount
```

Рисунок 14

В Python функції можуть повертати кілька значень одночасно:

```
def modifyName2(userName):
    newName1=userName.upper()
    newName2=userName.lower()
    return newName1, newName2

name=input("Input your name:")
print(modifyName2(name))
```

Результат роботи такої функції — це кортеж:

```
Input your name:Admin
('ADMIN', 'admin')
```

Рисунок 15

Для подальшої роботи, результати такої функції зручно присвоїти кільком змінним:

```
def modifyName2(userName):
    newName1=userName.upper()
    newName2=userName.lower()
    return newName1, newName2

name=input("Input your name:")
nameUpper, nameLower=modifyName2(name)
```

```
print(nameUpper)
print(nameLower)
```

```
Input your name:Admin
ADMIN
admin
```

Рисунок 16

Змінній, яка вказана першою у рядку `nameUpper`, `nameLower=modifyName2(name)` буде надано перший результат, який повертається за допомогою функції, тобто значення першої змінної після команди `return` у тілі функції.

Всередині тіла функції може бути декілька операторів `return`, але в процесі її роботи буде виконано лише один з них:

```
def checkLog(userLog):
    if userLog=='admin':
        return userLog.upper()
    elif userLog=='user':
        return userLog.lower()
    else:
        return "Wrong login!"

print(checkLog("admin"))
print(checkLog("user"))
print(checkLog("student"))
```

```
ADMIN
user
Wrong login!
```

Рисунок 17



## 2. Розширені прийоми в роботі з функціями

Припустимо, нам необхідно написати функцію, яка обчислює середнє арифметичне довільної кількості чисел (своїх аргументів). Виходить, що заздалегідь нам невідомо, скільки вхідних параметрів потрібно вказати у її визначенні: три, чотири, а може й десять. У різних ситуаціях функція буде обробляти різну кількість вхідних даних, тобто, набір (масив параметрів), довжина якого наперед невідома.

Для вирішення цієї проблеми в Python передбачені такі підходи, як упакування і розпакування аргументів. Розглянемо синтаксис, процес та приклади пакування аргументів функції.

Загальний синтаксис:

```
def ім'я_функції(*ім'я_набору_параметрів):  
    тіло функції
```

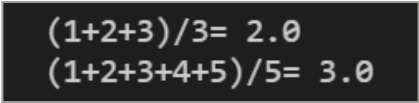
Обробка (процес пакування) відбувається таким чином: інтерпретатор у точці виклику функції «упаковує» (поміщає) всі аргументи (зазначені при виклику функції) у кортеж, ім'я якого наведено після символу «\*» у рядку оголошення функції. У тілі функції потрібно враховувати особливості роботи з кортежем (обробляти аргументи як елементи послідовності за допомогою їхньої позиції-індексу в послідовності). Тому такий параметр називають також «параметр із позиційними аргументами змінної довжини».

Ім'я параметра (набору аргументів) довільне (з урахуванням вимог до імен змінних), найчастіше використовують ім'я `*args`.

Розглянемо на прикладі.

```
def meanF(*args):
    s=0
    k=0
    for i in args:
        s+=i
        k+=1
    return s/k

print("(1+2+3)/3=", meanF(1,2,3))
print("(1+2+3+4+5)/5=", meanF(1,2,3,4,5))
```



```
(1+2+3)/3= 2.0
(1+2+3+4+5)/5= 3.0
```

Рисунок 18

Отже, ми маємо функцію, що працює з довільним числом аргументів. Однак, перераховувати весь набір аргументів при виклику функції (особливо, коли їх велика кількість) незручно і це також може стати причиною помилки в наборі при значній кількості значень.

Для вирішення цієї проблеми використовується розпакування аргументів функції — це спосіб передачі елементів колекції (списку, кортежу, словника) через кому на вхід функції як її аргументи.

Для цього ми спочатку створюємо таку колекцію:

```
numbers=[1,2,3,4,5]
```

А потім під час виклику функції передаємо їй цю колекцію як аргумент, поставивши перед її ім'ям символ «\*»:

```
print("(1+2+3+4+5)/5=", meanF(*numbers))
```

Розглянемо завдання: необхідно написати функцію, яка виводити логін та пароль користувача (кожен з них з нового рядка) згідно з деяким шаблоном.

Набір вихідних даних — це вкладений список (набір логінів та паролів кількох користувачів):

```
users=[['user1','111'],
        ['user2','2222'],
        ['user3','33333']]
```

Створимо нашу функцію:

```
def printInfo(*clients):

    for i in range(len(clients)):
        if i==0:
            print("login: {}".format(clients[i]))
        elif i==1:
            print("pass: {}".format(clients[i]))
```

Якщо ми просто обійдемо список у циклі, передаючи на кожному кроці вкладений список на вхід нашої функції, дані будуть виведені некоректно:

```
for user in users:
    printInfo(user)
```

```
login: ['user1', '111']  
login: ['user2', '2222']  
login: ['user3', '33333']
```

Рисунок 19

Використання підходу з розпакуванням вирішить проблему:

```
for user in users:  
    printInfo(*user)
```

```
login: user1  
pass: 111  
login: user2  
pass: 2222  
login: user3  
pass: 33333
```

Рисунок 20

## 2.1. Аргументи за замовчуванням, аргументи-ключі

Припустимо, у нас є функція, яка приймає два аргументи (логін та пароль користувача) і залежно від значення логіну видає різні вітальні повідомлення для користувача.

```
def userGreetings(login, passw):  
    if login=='admin':  
        print("Welcome, admin")  
    elif login=='student':  
        print("Welcome, student")
```

```

else:
    print("Welcome, guy")

userGreetings()

```

**Возникло исключение: TypeError** ×

`userGreetings()` missing 2 required positional arguments: 'login' and 'passw'

Рисунок 21

Якщо ми викличемо цю функцію, вказавши лише перший аргумент (логін):

```

userGreetings("admin")

```

то теж буде помилка:

**Возникло исключение: TypeError** ×

`userGreetings()` missing 1 required positional argument: 'passw'

Рисунок 22

Причина помилки зрозуміла: у виклику функції `userGreetings()` відсутні два необхідні аргументи або один із двох необхідних аргументів. Тобто нам потрібно, щоб та сама функція працювала і з двома, і з одним, і взагалі без аргументів? Створювати ще дві копії цієї функції, але з різним числом параметрів при оголошенні — не найраціональніший варіант вирішення цієї проблеми.

Наприклад, якщо в системі є пароль по замовчуванню для всіх користувачів «111» і кожен може зайти в систему або за допомогою свого пароля, або, використовуючи «111», то нам потрібно, щоб наша функція `userGreetings()` працювала і з двома аргументами, і з одним (логіном).

Для обробки таких ситуацій передбачено параметри (аргументи) за замовчуванням. У функціях аргумент може бути обов'язковим (як у наших попередніх прикладах) і необов'язковим. У необов'язкового аргумента значення повинно бути задано по замовчуванню.

Обробка таких ситуацій відрізняється лише у рядку оголошення функції:

```
def userGreetings(login, passw="111") :  
    if login=='admin':  
        print("Welcome, admin")  
    elif login=='student':  
        print("Welcome, student")  
    else:  
        print("Welcome, guy")
```

Перший аргумент (обов'язковий) наведено стандартним способом, а ось другий (необов'язковий) — це аргумент зі значенням за замовчуванням, тому він наводиться у такому форматі:

```
ім'я_аргументу = значення_аргументу
```

Рядок із викликом функції в цьому випадку може містити як один, так і два аргументи:

```
userGreetings("admin", "12345")  
userGreetings("student")
```

```
Welcome, admin  
Welcome, student
```

Рисунок 23

Ми можемо створювати аргументи за замовчуванням будь-якої кількості (залежно від умови завдання), але всі вони мають бути наведені в кінці списку аргументів функції після обов'язкових аргументів (якщо такі передбачені).

Розглянемо синтаксично правильні та неправильні рядки коду для оголошення функції з аргументами за замовчуванням:

```
def childParent(childName = 'Bob', parent):
```

невірно

```
def childParent(parent, childName='Bob'):
```

вірно

```
def checkUser(userName, userLog = 'student',
               userPass='111'):
```

вірно

```
def checkUser(userName, userLog = 'student',
               userPass):
```

невірно

У попередніх ситуаціях ми працювали з позиційними аргументами функцій (найпоширеніший підхід), тобто, це такі аргументи, чиї значення у момент виклику функції копіюються у відповідні параметри функції, згідно з порядком їх розташування, у рядку оголошення функції.

Наприклад:

```
def myCalculation(a,b,c):
    return (a+b)**c
def myCalculation1(a,b,c=50):
    return (a+b)**c

print(myCalculation(10,20,30)) #a=10, b=20, c=30
print(myCalculation1(10,20)) #a=10, b=20, c=50
```

Незважаючи на популярність використання позиційних аргументів, у них є наступний недолік: потрібно пам'ятати порядок (позицію) кожного аргументу в оголошенні функції, щоб правильно вказати потрібні значення (або змінні) при виклику функції. Наприклад, у нас є функція, яка обчислює індекс маси тіла:

$$I = \frac{m}{h^2},$$

де:

**m** — маса тіла у кілограмах;

**h** — зріст в метрах.

```
def calculationBMI(m,h):
    return m/(h**2)

print(calculationBMI(50,1.6)) #19.53
```

У рядку виклику функції ми просто вказуємо потрібні значення (зросту, ваги), але якщо ми випадково їх переплутаємо, то отримаємо неправильне значення індексу маси тіла.

```
print(calculationBMI(1.6, 50)) #0.00064
```

Щоб уникнути подібної плутанини в Python передбачені аргументи-ключі (keyword-аргументи). Їх використання відбувається у точці виклику функції:

```
print(calculationBMI(h=1.6, m=50)) #19.53
```

При цьому потрібно використовувати такі самі імена аргументів, як і в оголошенні функції. Проте, порядок їх слідування може бути будь-яким.



Можна в одній функції використовувати і позиційні, і keyword-аргументи. Важливо пам'ятати, що першими (у виклику функції) мають бути позиційні аргументи:

```
def calculationUserBMI(m,h,name):
    print ("{}'s BMI={}".format(name, m/(h**2)))

calculationUserBMI(50,name='Jane',h=1.6) #19.53
```

## 2.2. Область видимості, правило LEGB

Концепція змінних, яка дозволяє зберігати значення, а потім витягувати і (за потреби) змінювати ці значення, є однією з основних у всіх мовах програмування. Коли ми створюємо змінні у програмах, потрібно чітко розуміти «хто їх може бачити», тобто в якій частині програми їх можна знайти та отримати їхнє значення. Ці правила визначають межі кожної змінної: в якому місці програми вона доступна.

Область видимості змінних — це така частина програмного коду, в рамках якої на ім'я змінної (ідентифікатору) можна отримати (змінити) значення. Поза областю видимості цей ідентифікатор може бути вільний або пов'язаний абсолютно з іншим значенням.

Основне призначення області видимості — це запобігання конфліктів ідентифікаторів і, як наслідок, непередбачуваної поведінки програми.

Найпоширенішими і найчастіше використовуваними у багатьох мовах програмування є:

- **глобальна область видимості** — ідентифікатори, що визначаються у цій області, доступні у будь-якому місці програми;

- **локальна область видимості** — ідентифікатори, що визначаються в цій області, доступні тільки для коду в цій області (наприклад, в межах конкретної функції).

Оскільки Python є мовою з динамічною типізацією, то змінні створюються тоді, коли ми вперше надаємо їм деяке значення. На основі цієї операції присвоєння інтерпретатор Python визначає область видимості даного ідентифікатора, тобто та область коду, де створюємо змінну (функцію), визначає її область видимості.

Кожного разу, коли ми в програмі використовуємо певний ідентифікатор (ім'я змінної або функції), інтерпретатор Python проходить по всіх рівнях області видимості для пошуку такого ідентифікатора. Якщо такий ідентифікатор існує, то ми отримуємо результат його першого виявлення (у випадках, коли один ідентифікатор використовується в різних областях видимості).

**Built-in** – вбудована області видимості на рівні мови Python

**Global** – глобальна області видимості на рівні модуля

**Enclosing** – вложенная области видимости «охватывающих» функций

**Local** – локальная области видимости внутри функций

Рисунок 24

Цей процес пошуку слідує правилу LEGB (*Local* — локальна, *Enclosing* — вкладена, *Global* — глобальна і *Built-in* — вбудована) (рис. 24).

**Локальна область видимості** (*Local*) — це частина коду, яка відповідає тілу деякої функції або лямбда-вираженню (розглянемо в уроках далі). Ідентифікатори, визначені всередині функції, відносяться до її локальної області видимості та існують тільки в рамках цієї функції, або її локальної області. Ця область створюється не при оголошенні функції, а при її виклику (тобто скільки було викликів функції, стільки і різних, нових, локальних областей).

Після завершення роботи функції локальна область видимості знищується, а ідентифікатори «забуваються».

**Вкладена (нелокальна) область видимості** (*Enclosing*) — дана область видимості виникає під час роботи вкладених функцій і містить ідентифікатори, які ми визначаємо у вкладеній функції. Ідентифікатори цієї області «помітні» з коду внутрішніх та «охоплюючих» функцій (всередині яких є інші внутрішні функції).

**Глобальна область видимості** (*Global*) — це рівень програми (скрипту, модуля) Python, який містить усі ідентифікатори, які ми визначили на рівні програми (тобто поза функціями, якщо вони є). Ця область створюється під час запуску нашої програми. Ідентифікатори, визначені в цій області, будуть доступні в будь-якому місці програмного коду. Під час виконання програми існує єдина глобальна область видимості, яка зберігається доки наша програма не завершиться, після чого всі ідентифікатори цієї області будуть забуті.

**Вбудована область видимості** (*Built-in*) — це рівень мови Python, тобто, всі вбудовані об'єкти, функції Python знаходяться у цій області. Ця область активується в момент запуску інтерпретатора Python і всі вбудовані об'єкти автоматично завантажуються до неї. Таким чином, ми можемо застосовувати їх (наприклад, вбудовану функцію `abs()`) без використання операції імпорту будь-якого модуля.

### 2.3. Локальні та глобальні змінні у функціях

Як ми знаємо, у момент виклику функції створюється нова локальна область видимості.

Усі ідентифікатори, призначені всередині функції, створюються в момент її виклику, видимі (доступні) тільки в межах цієї функції (всередині інструкції `def`) і будуть знищені, коли робота функції закінчиться.

Розглянемо з прикладу:

```
def calculateExample1():
    baseVar=int(input("input base variable: "))
    resultVar = baseVar ** 2
    print(f'The square of {baseVar} is: {resultVar}')

calculateExample1()
```

Функція `calculateExample1()` обчислює квадрат введеного користувачем цілого числа.

Після першого виклику цієї функції створюється локальна область видимості, яка містить дві змінні: `baseVar` зі значенням 3 (оскільки користувач у нашому випадку ввів значення 3) і `resultVar`, яка після обчислення отримає значення 9.

```
input base variable: 3
The square of 3 is: 9
```

Рисунок 25

Спробуймо викликати нашу функцію ще раз:

```
def calculateExample1():
    baseVar=int(input("input base variable: "))
    resultVar = baseVar ** 2
    print(f'The square of {baseVar} is: {resultVar}')

calculateExample1()
calculateExample1()
```

Спочатку створюється перша область видимості (після першого виклику функції `calculateExample1()`), в якій змінна `baseVar` дорівнює 4 (припускаємо, що користувач ввів значення 4), а змінна `resultVar` отримує значення 16. Після закінчення роботи функції (після її першого виклику) ці змінні зі своїми значеннями зникають (тобто програма їх не запам'ятовує)

При другому виклику функції `calculateExample1()` створиться нова локальна область з іншими локальними змінними `baseVar` та `resultVar` (тобто відбудеться нове виділення нових фрагментів пам'яті для них) зі значеннями 5 (припускаємо, що користувач вдруте ввів значення 5) і 25.

```
input base variable: 4
The square of 4 is: 16
input base variable: 5
The square of 5 is: 25
```

Рисунок 26

Якщо ми спробуємо звернутися до змінної `baseVar` (або `baseVar`) поза функцією (наприклад, після її виклику),

```
def calculateExample1():
    baseVar=int(input("input base variable: "))
    resultVar = baseVar ** 2
    print(f'The square of {baseVar} is: {resultVar}')

calculateExample1()
print(baseVar)
```

то отримаємо помилку `NameError`, тому що дані змінні існують лише у локальній області, яка, в свою чергу, існує лише у момент виклику (роботи) функції `calculateExample1()`.

**name 'baseVar' is not defined**

Рисунок 27

У повідомленні про помилку `NameError` буде вказано ім'я (у нашому прикладі `baseVar`), яке інтерпретатору не вдалося виявити.

Оскільки у нас немає доступу до локальних змінних за межами функцій, які їх створили, отже в різних функціях можна використовувати однакові імена змінних.

Наприклад:

```
def calculateExample1():
    baseVar=int(input("input base variable: "))
    resultVar = baseVar ** 2
    print(f'The square of {baseVar} is: {resultVar}')

def calculateExample2():
    baseVar=int(input("input base variable: "))
```

```

resultVar = baseVar ** 3 + 10
print(f'The result is: {resultVar}')

calculateExample1()
calculateExample2()

```

Наша друга функція `calculateExample2()` використовує у своєму тілі змінні з такими ж іменами, як і перша функція `calculateExample1()`. Функція `calculateExample2()` не може бачити локальні змінні функції `calculateExample1()` і навпаки. Тому обидві функції працюють коректно і конфлікту імен змінних не виникає.

```

input base variable: 4
The square of 4 is: 16
input base variable: 2
The result is: 18

```

Рисунок 28

Як було розглянуто раніше, з моменту запуску нашої програми утворюється глобальна область видимості і всі ідентифікатори, визначені в ній (за межами функцій), доступні з будь-якої точки коду, в тому числі й всередині функцій.

Розглянемо приклад:

```

userName = "Bob"

def checkUser():
    userLog=input("Input your login: ")
    if userLog=='admin':

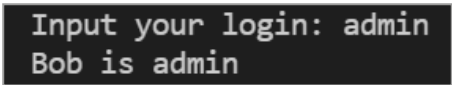
```

```

        print("{} is admin".format(userName))
    else:
        print("{} is user".format(userName))
checkUser()

```

Ми маємо глобальну змінну `userName`, яку ми використовуємо (зчитуємо її значення) всередині нашої функції `checkUser()` для виведення повідомлення для користувача:



```

Input your login: admin
Bob is admin

```

Рисунок 29

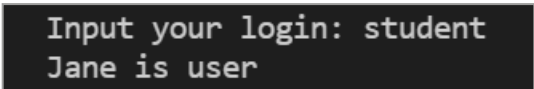
Тепер створимо всередині цієї функції локальну змінну з таким самим ім'ям `userName` і надамо їй інше значення.

```

userName="Bob"
def checkUser():
    userLog=input("Input your login: ")
    userName="Jane"
    if userLog=='admin':
        print("{} is admin".format(userName))
    else:
        print("{} is user".format(userName))
checkUser()

```

У цьому випадку, при зверненні до змінної `userName`, буде використано локальну змінну з її значенням «*Jane*»:



```

Input your login: student
Jane is user

```

Рисунок 30



Для того, щоб всередині функції змінити значення глобальної змінної, необхідно використовувати ключове слово **global**. Використовуючи **global** ми можемо визначити ідентифікатор (або список ідентифікаторів), які наша програма сприйматиме як глобальні.

Розглянемо з прикладу:

```
userName="Bob"
print("{} - first user".format(userName))

def checkUser():
    global userName
    userName="Joe"
    userLog=input("Input your login: ")
    if userLog=='admin':
        print("{} is admin".format(userName))
    else:
        print("{} is user".format(userName))

checkUser()
print("{} - second user".format(userName))
```

Всередині нашої функції **checkUser()** ми вказуємо інтерпретатору рядком коду **global userName**, що цю змінну потрібно шукати в глобальній області видимості, а не створювати нове локальне ім'я. В результаті існуюче значення змінної **userName** було змінено всередині функції на нове.

```
Bob - first user
Input your login: admin
Joe is admin
Joe - second user
```

Рисунок 31

Використання глобальних змінних має серйозний недолік: їх значення можуть змінюватися всередині функцій. Цю зміну ми можемо не помітити, припускаючи, наприклад, що наші змінні зберігають початкові значення.

Розглянемо наступний приклад: припустимо, що нам необхідно написати функцію авторизації користувача, яка приймає на вхід лише пароль. Логін є глобальною змінною, значення якої потрібно лише зчитувати в тілі функції для перевірки відповідності пароля.

При цьому програмістом у тілі функції була викликана інша функція, що виводить привітання для користувача. Припустимо, що програміст не бачив її код (всередині якого, можливо, випадково відкритий доступ на зміну цієї змінної `userLog` за допомогою оператора `global` і далі їй присвоєно значення «*admin*») і просто скористався нею для виведення вітання в процесі авторизації.

```
def greetingUser():
    global userLog
    userLog="admin"
    print("Welcome!")

def checkPassw(passw):
    greetingUser()
    if userLog=="admin" and passw=='111':
        print("{} , access is allowed. You are admin!".
              format(userLog))
    elif userLog=="student" and passw=='222':
        print("{} , access is allowed. Welcome!".
              format(userLog))
    else:
        print("{} , access isn't allowed.
              Wrong password!".format(userLog))
```

```
userLog=input("Input your login: ")
userPassw=input("Input your password: ")

checkPassw(userPassw)
```

В результаті, коли користувач ввів значення логіна «*student*» і воно потрапило в глобальну змінну `userLog`, то далі після виклику функції `checkPassw()`, введене значення «*student*» було всередині функції змінено на «*admin*». Після цього процес авторизації пройшов некоректно незважаючи на те, що користувач ввів коректні значення логіну та пароля.

```
Input your login: student
Input your password: 222
Welcome!
admin, access isn't allowed. Wrong password!
```

Рисунок 32

Таким чином, глобальні змінні потенційно дозволяють функціям мати неявні, не завжди легко і відразу знайдені, побічні ефекти.

## 2.4. Функції як об'єкти першого класу

В багатьох мовах програмування існує поняття об'єкт першого класу.

Об'єкт першого класу — це такий об'єкт, що створюється динамічно (під час виконання програми). Його можна передавати функції як аргумент і повертати як значення, одержане в результаті роботи функції. Також

об'єкт першого класу можна присвоїти певній змінній як значення. Таким чином, функції Python є об'єктами першого класу.

В Python функції є об'єктами. Використаємо вбудовану функцію `type()`, щоб вивести тип даних нашої функції `sayHello()`:

```
def sayHello():
    name=input("What's is your name? ")
    print("Hello, {}".format(name))

print(type(sayHello)) #<class 'function'>
```

Тепер давайте присвоїмо функцію як значення змінної і викличемо її за допомогою цієї змінної:

```
def sayHello():
    name=input("What's is your name? ")
    print("Hello, {}".format(name))

greeting=sayHello
greeting()
```

```
What's is your name? Student
Hello, Student
```

Рисунок 33

У нашому прикладі функція `sayHello` присвоюється змінною `greeting` як значення. Зверніть увагу, що присвоюється тільки ім'я `sayHello` без наступних круглих дужок, тому що змінна `greeting` повинна отримати посилання на об'єкт функції.

Далі, вказавши після імені змінної `greeting` круглі дужки, ми запустили ту функцію, на яку ця змінна посиляється. Тепер розглянемо, як можна передати функцію як аргумент іншої функції.

Припустимо, у нас є список клієнтів і є функція, яка виводить різні привітання для одного клієнта, залежно від того, чи є він адміністратором, студентом чи просто клієнтом.

```
customers=['AdminJane', 'adminBob', 'STUDENTbob',
           'studentAlice', 'Kate']

def sayHello(customer):
    if customer.find("admin")!=-1:
        print("Hello, admin - {}".format(customer))
    elif customer.find("student")!=-1:
        print("Hello, student - {}".format(customer))
    else:
        print("Hello, {}".format(customer))
```

Для коректної роботи нам необхідно перед запуском нашої функції `sayHello(customer)` перевести логін клієнта в нижній регістр (бо всередині функції в перевірках логінів використовується нижній регістр).

Реалізуємо це окремою функцією, яка прийматиме на вхід список логінів клієнтів і для кожного з них (після попереднього переведення в нижній регістр) запускати функцію `sayHello(customer)`:

```
customers=['AdminJane', 'adminBob', 'STUDENTbob',
           'studentAlice', 'Kate']

def sayHello(customer):
```

```

if customer.find("admin")!=-1:
    print("Hello, admin - {}".format(customer))
elif customer.find("student")!=-1:
    print("Hello, student - {}".format(customer))
else:
    print("Hello, {}".format(customer))

def greetings(customList, greetF):
    for c in customList:
        greetF(c.lower())

greetings(customers, sayHello)

```

Результат:

```

Hello, admin - adminjane
Hello, admin - adminbob
Hello, student - studentbob
Hello, student - studentalice
Hello, kate

```

Рисунок 34

Наступне наше питання — як повертати функцію як значення і коли нам це може знадобитися?

Припустимо, що ми маємо прості три функції для обчислення різних виразів:

```

def calculateExample1(a,b):
    return a**b+2*a-b/0.23

def calculateExample2(a,b):
    return a*b-5**a+b

```

```
def calculateExample3(a,b):
    return b**4+a**2-a/b
```

Користувач вибирає, який із виразів йому треба обчислити і ми маємо запустити потрібну функцію. Реалізуємо перевірку вибору користувача за допомогою окремої функції `userChoice()`, яка, у свою чергу, повертатиме функцію, яка виконує потрібні обчислення.

```
def userChoice(c):
    if c=="1":
        return calculateExample1
    elif c=="2":
        return calculateExample2
    elif c=="3":
        return calculateExample3
```

Тепер потрібно викликати нашу функцію `userChoice()` з аргументом, який відповідає вибору користувача та призначити результат її роботи (одну з функцій-обчислювачів) в окрему змінну `myCalculation`.

Потім, використовуючи змінну `myCalculation`, яка містить посилання на об'єкт потрібної функції-обчислювача, запустимо обчислення з аргументами `x` і `y`.

```
x=2
y=3

for i in range(3):
    userAncw=input("Your choice:")
    myCalculation = userChoice(userAncw)
    print(myCalculation(x,y))
```

Результат:

```

Your choice:1
-1.0434782608695645
Your choice:3
84.33333333333333
Your choice:2
-16

```

Рисунок 35

## 2.5. Рекурсія

Як ми вже знаємо, функції можуть викликати інші функції (у своєму тілі містити їх виклик). Також функції можуть викликати самі себе. Така властивість називається рекурсією, а функції рекурсивними. У програмуванні рекурсія дозволяє описувати (реалізовувати) повторюваний процес без використання операторів циклу.

Класичним прикладом, коли корисно використовувати рекурсію у програмуванні, є завдання обчислення факторіалу числа  $n!$ . Факторіал числа  $n$  — це добуток всіх чисел від 1 до  $n$  включно.

Формула для обчислення  $n!$ :

$$n! = 1 * 2 * 3 * \dots * n.$$

Наприклад,

$$3! = 1 * 2 * 3$$

$$4! = 1 * 2 * 3 * 4 = 3! * 4$$

$$5! = 1 * 2 * 3 * 4 * 5 = 4! * 5$$

При цьому:  $0!=1$ . Таким чином, можна описати процес обчислення факторіалу так:



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

Наприклад,

$3! = 3 \cdot 2!$  — нам потрібно вирахувати  $2!$

$2! = 2 \cdot 1!$  — нам потрібно вирахувати  $1!$

$1! = 1 \cdot 0!$  — нам потрібно вирахувати  $0!$

$0! = 1$  — отримано результат.

Тепер починаємо «підніматися»: передаємо значення  $1$  на «рівень» вище:  $1 \cdot 1 = 1$ . Передаємо це обчислене значення  $1$  на наступний «рівень» зверху:  $2 \cdot 1 = 2$ . Знову передаємо це обчислене значення  $2$  на наступний «рівень» зверху:  $3 \cdot 2 = 6$  — підсумковий результат.

Тобто, щоб обчислити  $3!$ , нам треба спочатку обчислити  $2!$ , і ще раніше обчислити  $1!$  і т. п.

Зручно процедуру обчислення факторіалу представити у вигляді функції, яка за умови  $n > 0$ , буде викликати саму себе з аргументом  $(n-1)$ , а при  $n=0$  поверне значення  $1$ .

```
def factorialCalculation(n):
    if n==0:
        return 1
    else:
        return n*factorialCalculation(n-1)
print(factorialCalculation(3))
```

Після запуску нашої функції відбуватиметься наступне:

$3 \cdot \text{factorialCalculation}(2)$

$2 \cdot \text{factorialCalculation}(1)$

$1 \cdot \text{factorialCalculation}(0)$

$1$

Рекурсивна функція повинна обов'язково містити хоча б один оператор **return**, який повертає звичайне значення (точка завершення рекурсії), щоб не допустити нескінченне виконання програми (зациклення).

Розглянемо наступне завдання: необхідно перевірити, чи є слово паліндромом (тобто читається у прямому та зворотному напрямку однаково, наприклад, слово «мадам» — паліндром).

```
def isStrPalind(myStr):
    if len(myStr) == 0:
        return True
    else:
        if myStr[0] == myStr[-1]:
            return isStrPalind(myStr[1:-1])
        else:
            return False

userStr = input("Your word: ")
if (isStrPalind(userStr.lower()) == True):
    print("{} is palindrome".format(userStr))
else:
    print("{} isn't palindrome".format(userStr))
```

Результат:

```
Your word: madam
madam is palindrome
```

Рисунок 36

```
Your word: Level
Level is palindrome
```

Рисунок 37

```
Your word: music
music isn't palindrome
```

Рисунок 38

Рішення полягає в послідовному порівнянні першого і останнього символу рядка `myStr`, потім другого та передостаннього символів (тобто знову першого і останнього символу підрядка, отриманого з вихідного зрізу `myStr[1:-1]`) і т. д. При цьому подальша перевірка підрядка, що отримується з поточної, може запускатися, тільки якщо дорівнені символи збігаються, інакше рекурсія зупиняється, і ми отримуємо результат `False`.

Як умову для зупинки рекурсії використовуємо значення довжини рядка, що дорівнює нулю (усі символи рядка перевірені). В даному випадку отримуємо результат `True`.

Розглянемо ще один приклад використання рекурсії. Припустимо, що є список чисел, і нам необхідно написати функцію, яка знайде мінімальне число у цьому списку. Ми знаємо, що в Python є вбудована функція `min()`, яка знаходить мінімум серед двох чисел, тобто можна процес пошуку представити у вигляді процесу порівняння двох повторюваних чисел.

Починаємо з кінця списку. Відокремлюємо останній елемент та запускаємо пошук на зрізі без останнього елемента. На другому кроці відокремлюємо передостанній елемент і запускаємо пошук на зрізі вже без двох елементів і т. д.

На останньому кроці в зрізі залишиться лише один елемент (перший елемент початкового списку). Відомо, що й у списку одне число, тоді результат (мінімум) — це число. Це буде точкою зупинки нашої рекурсії, і функція

почне «підніматися вгору», передаючи отриманий результат на верхній рівень. У нашому прикладі перший результат ми отримаємо на п'ятому рівні (це число 2), який буде переданий на четвертий, де буде знайдено мінімум між цим числом 2 і «очікуваним» повернення рекурсії числом 4. Знайдений мінімум (2) буде передано на третій рівень, де прирівнюватиметься з черговим «очікуваним» поверненням рекурсії числом 1 і т. п.

2	4	1	8	3		
					min(1, 3) ?	1
	2	4	1	8		
					min(1, 8) ?	1
		2	4	1		
					min(2, 1) ?	1
			2	4		
					min(2, 4) ?	2
				2		

Рисунок 39

```
def findMin(numberList):
    if len(numberList)>1:
        return min(findMin(numberList[:-1]),
                    numberList[-1])
    else:
        return numberList[0]
myNumbers=[2,4,1,8,3]
print("min={}".format(findMin(myNumbers))) #min=1
```

Основним недоліком рекурсії є те, що при кожному запуску функції (рекурсивний виклик) створюється копія параметрів функції та її локальних змінних, що викликає додаткові витрати пам'яті та процесорного часу.

## 3. Функціональне програмування

### 3.1. Що таке функціональне програмування?

Функціональне програмування — це такий підхід до розробки програмного забезпечення, при якому основна логіка задачі (алгоритм) представляється у вигляді набору функцій, що реалізують окремі кроки загального алгоритму.

Наприклад, завдання знаходження успішних студентів у групі (з середнім балом понад 60) зручно розбити на такі кроки (окремі підзадачі):

- введення даних про оцінки студентів групи;
- підрахунок середнього балу кожного студента з усіх предметів;
- знаходження студентів із середнім балом понад 60 та формування списку успішних студентів;
- виведення списку успішних студентів з їх середнім балом.

Кожне з цих завдань легко реалізувати та просто тестувати у вигляді окремої функції.

Як ми вже знаємо, функції можуть бути легко викликані та повторно використані за потреби у будь-якій точці загальної програми. Це забезпечує легкість у керуванні кодом. Також роботу над проектом можуть вести паралельно декілька розробників, кожен з яких реалізовуватиме свою частину логіки (у вигляді окремих функцій).

Python використовує концепцію функціонального програмування, надаючи нам великий набір вбудованих функцій. Також ми маємо можливість застосовувати особливості функції, як об'єктів першого класу, передавати їх в інші функції як параметри.

Функціональне програмування передбачає розробку з використанням чистих функцій.

Чиста функція — це така функція, яка реалізує зв'язок між вхідними та вихідними даними (забезпечує перетворення вхідних значень у вихідні за деяким алгоритмом).

Чиста функція:

- завжди видає однакову відповідь при тих самих вхідних параметрах;
- немає прихованого (чи неявного) введення-виведення чи інших побічних ефектів, тобто працює тільки із вхідними значеннями (параметрами) без участі зовнішніх даних.

Завдяки цим властивостям у програмах, створених з використанням методик функціонального програмування, легко виявляються та виправляються помилки.

Окрім цієї переваги функціональна парадигма популярна в програмуванні також через високий рівень абстракції: ми описуємо за допомогою викликів функцій потрібний результат, а не кроки по його досягненню. Таким чином, код основної програми стає коротшим, але сенс кожного рядка прозоріший.

Додаткову прозорість коду також забезпечують і найчистіші функції, які виключають можливість побічних ефектів і проміжних значень.

Значними у функціональному програмуванні є можливості функцій як об'єктів першого класу (з якими ми вже знайомі): функція може використовувати іншу функцію, як свій аргумент, і може повернути іншу функцію, як результат своєї роботи (викликаючій стороні, наприклад, в основну програму).

Давайте пригадаємо, як це працює: присвоїмо функцію змінній (створивши нове посилання на цей же код функції, але з іншим ім'ям) і використаємо цю змінну для виклику функції:

```
def myGreeting1():
    print("Good morning! Have a nice day!")

sayGoodMorning=myGreeting1
sayGoodMorning()
```

Ми можемо “збільшити масштаб” цієї ідеї: помістимо імена функцій у список та у циклі викличемо кожену:

```
def myGreeting1():
    print("Good morning! Have a nice day!")

def myGreeting2():
    print("Good day! Nice to see you!")

def myGreeting3():
    print("Hey! Long-time no see.")

def myGreeting4():
    print("Good night! See you tomorrow.")

myGreetingsList=(myGreeting1, myGreeting2,
                  myGreeting3, myGreeting4)
```

```
for myGreeting in myGreetingsList:
    myGreeting()
```

Результат:

```
Good morning! Have a nice day!
Good day! Nice to see you!
Hey! Long-time no see.
Good night! See you tomorrow.
```

Рисунок 40

Тепер передамо функцію, як аргумент іншої функції, щоб запитати ім'я користувача, якому призначено привітання:

```
def greetingRecipient(greetFunction):
    greetRecipient=input("name?")
    print("Dear, ", greetRecipient)

    greetFunction()
```

І привітаємо кожного користувача по-різному, запустивши нашу функцію `greetingRecipient()` у циклі над списком функцій — привітань:

```
for myGreeting in myGreetingsList:
    greetingRecipient(myGreeting)
```

Результат на рисунку 41.

Такий підхід називається функціональною композицією. Так само, як ми передавали функцію для іншої функції як аргумент, ми можемо повернути функцію в якості значення, що повертається.



```

name?Hanna
Dear, Hanna
Good morning! Have a nice day!
name?Joe
Dear, Joe
Good day! Nice to see you!
name?Bob
Dear, Bob
Hey! Long-time no see.
name?Jane
Dear, Jane
Good night! See you tomorrow.

```

Рисунок 41

Створимо функцію, яка повертає потрібну функцію із списку залежно від коду часу доби:

```

def checkTimeOfDay():
    while True:
        timeOfDay=input("Input time of day (M-morning;
                        D-afternoon;E- everning;N-night):")
        if timeOfDay=="M":
            return myGreetingsList[0]
        elif timeOfDay=="D":
            return myGreetingsList[1]
        elif timeOfDay=="E":
            return myGreetingsList[2]
        elif timeOfDay=="N":
            return myGreetingsList[3]
        else:
            print("Wrong input!")

```

Тепер можна в циклі запустити тестування, наприклад, для чотирьох користувачів. Запитаємо їх імена та код часу доби, просто викликавши нашу функцію

`greetingRecipient()` і передавши їй як аргумент виклику функції `checkTimeOfDay()`:

```
for i in range(3):
    greetingRecipient(checkTimeOfDay())
```

Результат:

```
Input time of day (M-morning;D-afternoon;E- evening;N-night):0
Wrong input!
Input time of day (M-morning;D-afternoon;E- evening;N-night):M
name?Hanna
Dear, Hanna
Good morning! Have a nice day!
Input time of day (M-morning;D-afternoon;E- evening;N-night):D
name?Bob
Dear, Bob
Good day! Nice to see you!
Input time of day (M-morning;D-afternoon;E- evening;N-night):E
name?Jane
Dear, Jane
Hey! Long-time no see.
```

Рисунок 42

Таким чином, Python надає нам все необхідне для використання парадигми функціонального програмування.

### 3.2. Анонімні функції `lambda`

Ми вже знаємо, як створювати функції за допомогою ключового слова `def`. Але, крім цього підходу, Python дозволяє створювати функції у вигляді виразів.

У цьому випадку використовується ключове слово `lambda` з таким синтаксисом оголошення `lambda`-функцій:

```
lambda аргумент1, аргумент2, ..., аргументN: вираз
```

- аргумент1, аргумент2, ..., аргументN — імена змінних-аргументів, які будуть використовуватись у виразі;
- вираз — сам вираз, який реалізовує деяке обчислення (чи дію).

Наприклад, нам потрібно створити просту функцію, яка додає 10 до деякого значення. Звичайно, можна реалізувати цю функцію за допомогою ключового слова `def` (як ми робили раніше):

```
def add10(x):
    return x+10
```

Припустимо, що ми маємо список чисел, які потрібно перетворити за допомогою цієї функції:

```
myNumbers=[2, 2.5, 4.56,23]

for num in myNumbers:
    print(add10(num))
```

Фактично, наша функція містить лише один вираз всередині. Найзручніше тут використовувати лямбда-функцію:

```
myNumbers=[2, 2.5, 4.56,23]

newNum=lambda x:x+10

for num in myNumbers:
    print(newNum(num))
```

Також можна створювати лямбда-функцію саме в тому ж місці, де вона буде використана:

```
myNumbers=[2, 2.5, 4.56,23]

for num in myNumbers:
    print((lambda x:x+10)(num))
```

Особливо корисним такий підхід є у ситуаціях, коли надалі не передбачається використання цієї функції.

Проаналізувавши загальний синтаксис `lambda`-виразу та приклади вище, можна виділити такі відмінності `lambda`-функцій від `def`-підходу:

- `lambda`-вирази записуються в один рядок (при спробі розмістити вираз у другому рядку ми отримаємо помилку);
- `lambda`-вирази створюють функції, у яких немає імені (анонімні функції), які можна присвоїти змінній (як у прикладі вище) або використовувати саме в місці створення;
- `lambda`-функція містить лише один вираз, результат обчислення якого — це результат, який повертається `lambda`-функцією, тобто ключове слово `return` не потрібно використовувати;
- `lambda`-функції призначені для реалізації (обчислення) більш простих фрагментів коду в порівнянні зі звичайними функціями.

Оскільки `lambda`-функції є виразами, значить, їх можна використовувати там, де використання `def` не допускається — всередині літералів або всередині виклику функцій.

Розглянемо наступний приклад: у нас є список студентів із середніми балами, причому інформація про

кожного студента також є списком, першим елементом якого є ім'я студента, а другим — його середній бал.

Нам необхідно відсортувати список студентів за іменами. Для сортування будемо використовувати вбудовану функцію `sorted`, першим аргументом якої є наш список, а другим — ключ сортування (ім'я студента у нашому завданні).

Для задання ключа сортування нам потрібно виділити із внутрішнього списку ім'я, що ми і зробимо за допомогою `lambda`-функції.

```
students = [['Bob', 70],
            ['Jane', 80],
            ['Andy', 50]]
print(students)
sortedSts=sorted(students, key=lambda x: x[1])
print(sortedSts)
```

Результат:

```
 [['Bob', 70], ['Jane', 80], ['Andy', 50]]
 [['Andy', 50], ['Bob', 70], ['Jane', 80]]
```

Рисунок 43

Тепер припустимо, що користувач вводить ціни двох товарів в одній валюті, а нам потрібно вивести їх в іншій валюті за курсом та врахувати знижку.

```
grnToDollar=28.2
discount=0.15
priceDol=lambda x:x/grnToDollar*(1-discount)
```

```
priceGrn1=float(input("Input price1 in grn:"))
print("price: {:.2f}$".format(priceDol(priceGrn1)))

priceGrn2=float(input("Input price2 in grn:"))
print("price: {:.2f}$".format(priceDol(priceGrn2)))
```

Результат:

```
Input price1 in grn:3000
price: 90.43$
Input price2 in grn:1245
price: 37.53$
```

Рисунок 44

Тут ми використовуємо ту саму `lambda`-функцію двічі, зберігши її як вираз у змінній `priceDol`, у яку далі передаємо різні ціни як аргументи.

Наше наступне завдання — реалізувати складання рядка з ім'ям та прізвищем користувача, які мають починатися з великої літери. Виконаємо її також із використанням `lambda`-функції, але вже з двома аргументами:

```
userName = lambda firstName, lastName:
    "Full user's name: {} {}".format(firstName.title(),
                                       lastName.title())

firstUserName=input("Input your first name:")
lastUserName=input("Input your last name:")

print(userName(firstName, lastUserName))
```

Результат:

```
Input your first name:joe  
Input your last name:BLACK  
Full user's name: Joe Black
```

Рисунок 45

На цих прикладах можна побачити ще одну перевагу використання `lambda`-функції — близькість програмного коду. Анонімні функції (код, який реалізує їхню логіку), розміщуються поруч у програмі, що підвищує читабельність коду та не вимагає використання додаткових імен функцій (як у випадку з використанням `def`).

Використовуючи `lambda`-функції, ми також зменшимо загальну кількість імен у нашій програмі, знижуючи кількість можливих ситуацій із конфліктами імен у цьому файлі коду.

### 3.3. Функції вищих порядків

Однією з найбільш важливих і корисних концепцій функціонального програмування є ідея функцій вищих порядків. Функція називається функцією вищого порядку, якщо вона приймає іншу функцію (функції) в якості аргументів та (або) повертає функцію в результаті своєї роботи.

Ми вже знаємо, що в Python функції — це об'єкти першого класу, тому можливість приймати як аргумент іншу функцію та/або повертати функцію як результат роботи активно використовується в Python-програмуванні.

Функції вищого порядку забезпечують більш гнучкий, легкий та компактний спосіб реалізації логіки програми за допомогою функцій.

Припустимо, що нам необхідно отримати вік студентів, чії роки народження зберігаються у списку.

Використовуючи підхід із застосуванням функцій вищих порядків, це завдання можна реалізувати таким чином:

```
studBirthYear = [2000, 1997, 2002, 1999, 2007]
studAges=list(map(lambda x:2022-x,studBirthYear))
print(studAges) # [22, 25, 20, 23, 15]
```

А без функцій вищих порядків код виглядатиме так:

```
studBirthYear = [2000, 1997, 2002, 1999, 2007]
studAges=[]

for year in studBirthYear:
    studAges.append(2022-year)

print(studAges) #[22, 25, 20, 23, 15]
```

У першому рішенні програмісту не потрібно заздалегідь створювати порожній список для зберігання віку, не потрібно згадувати, який метод об'єкта `list` (список) додається новий елемент в кінець списку, не потрібно навіть організовувати цикл.

Багато з розглянутих раніше вбудованих Python-функцій є функціями вищого порядку. Наприклад, вбудована функція `print()`. Нехай нам потрібно вивести модуль числа, що вводиться користувачем:

```
userNumber=int(input("Input your number:"))
print(abs(userNumber))
```



Тут вбудована функція `print()` приймає як аргумент іншу вбудовану функцію для отримання абсолютного значення числа — `abs()`, яка в свою чергу обробляє число користувача.

```
Input your number:-12
12
```

Рисунок 46

Вбудована функція `sorted()` також є функцією високого порядку. Ми можемо задавати ключ користувачького сортування за допомогою функції (наприклад, за допомогою тієї ж вбудованої функції `abs()` для отримання абсолютного значення числа):

```
myNumbers=[1.3, -2.3, -12, 11, 0.45]

print(myNumbers)
print(sorted(myNumbers, key=abs))
```

Результат:

```
[1.3, -2.3, -12, 11, 0.45]
[0.45, 1.3, -2.3, 11, -12]
```

Рисунок 47

Розглянемо ще один приклад: припустимо, нам необхідно написати функцію, яка в залежності від вибору користувача реалізує процес його аутентифікації, або зміну пароля, який призначений за замовчуванням, на новий. Дану функцію зручно реалізувати як функцію вищого порядку, назовемо її `userAction()`, яка запускатиме

(повертатиме) потрібну дію (функцію) для певного логіну та пароля користувача.

Для реалізації кожної з перерахованих можливостей створимо окремі функції (`userLogIn()`, `changePass()`), які перед виконанням потрібної дії перевіряють відповідність логіну і пароля.

```
adminPassw='111'
studentPassw='222'

def userLogIn(userLog, userPass):

    if (userLog.lower()=='admin') and
        (userPass==adminPassw):
        print("Welcome, admin")
    elif (userLog.lower()=='student') and
        (userPass==studentPassw):
        print("Welcome, student")
    else:
        print("Wrong data")

def changePass(userLog, userPass):
    global adminPassw
    global studentPassw
    if (userLog.lower()=='admin') and
        (userPass==adminPassw):
        adminPassw=input("Input new password for admin:")
    elif (userLog.lower()=='student') and
        (userPass==studentPassw):
        studentPassw=input("Input new password
                            for student:")
    else:
        print("Wrong data")

def userAction(login, passw, action):
    return action(login, passw)
```

```

userL=input("Login:")
userP=input("Password:")
userAnsw=input("1: LogIn; 2: Change password ")

if userAnsw=='1':
    userAction(userL, userP, userLogIn)
elif userAnsw=='2':
    userAction(userL, userP, changePass)
else:
    print("Wrong data")

```

Результати:

```

Login:Admin
Password:111
1: LogIn; 2: Change password 1
Welcome, admin

```

Рисунок 48

```

Login:STUDENT
Password:222
1: LogIn; 2: Change password 2
Input new password for student:333

```

Рисунок 49

Тепер припустимо, що нам потрібно реалізувати функцію, яка на основі імені користувача створює його логін (додаючи до імені рядок «**user**»). При цьому користувач може вибрати, в якому регістрі додавати його ім'я: у верхньому (наприклад, «**userkate**») або нижньому («**userKATE**»).

Для цього створимо дві функції для формування першого та другого варіанта логіну: **nameUpper(name)**, **nameLower(name)**. Також у нас буде функція вищого

порядку `makeLog(userName, maker)`, яка приймає ім'я користувача та функцію, яка його оброблятиме вибраним способом:

```
def nameUpper(name):
    return 'user'+name.upper()

def nameLower(name):
    return 'user'+name.lower()

def makeLog(userName, maker):
    return maker(userName)

user=input("Input your name:")
userAnsw=input("Select login-maker: 1-lower case;
               2 -upper case")

if userAnsw=='1':
    print(makeLog(user,nameLower))
elif userAnsw=='2':
    print(makeLog(user,nameUpper))
else:
    print("Wrong input!")
```

Результати:

```
Input your name:Bob
Select login-maker: 1-lower case;2 -upper case1
userbob
```

Рисунок 50

```
Input your name:jane
Select login-maker: 1-lower case;2 -upper case2
userJANE
```

Рисунок 51

### 3.4. Функції `map()`, `filter()`, `zip()`

У попередньому питанні ми розглянули поняття функцій вищих порядків, приклади створення власних функцій такого класу та кілька прикладів використання вбудованих функцій вищих порядків.

Однак, у всіх розглянутих випадках функції запускалися лише один раз з одним набором параметрів. А якщо нам необхідно запустити одну й ту ж функцію для кожного елемента деякого набору (наприклад, виконати якесь перетворення кожного елемента списку)? Або потрібно вибрати з набору елементи, що задовольняють певну умову?

Наприклад, у нас є список логінів користувачів:

```
userLogs=['123user45', 'USERstudent', '56use3', 'user-23',
          'adminUs']
```

Нам потрібно перевести кожен логін зі списку до нижнього регістру. Звичайно, це завдання можна виконати за допомогою операторів циклу та розгалуження. У нашому випадку, наприклад, таким чином:

```
userLogs=['123user45', 'USERstudent', '56use3',
          'user-23', 'adminUs']

userLogsLower=[]

for log in userLogs:
    userLogsLower.append(log.lower())

print(userLogsLower)
#['123user45', 'userstudent', '56use3', 'user-23',
  'adminus']
```

Оскільки перелічені завдання є дуже поширеними, то в Python передбачили вбудовані функції вищого порядку, що дозволяють реалізувати зазначену функціональність простим і компактним способом.

Вбудована функція `map()` дозволяє застосовувати потрібну функцію до кожного елемента колекції (об'єкта, що ітерується, наприклад, списку). Результат, що повертається — новий ітератор (об'єкт `map`), який потім можна передати, наприклад, у вбудовану функцію `list()` для створення списку результатів.

Загальний синтаксис функції `map()`:

```
map (function, iterable1, [iterable2,... iterableN])
```

`function` — функція-аргумент, в яку `map()` передає на «обробку» кожен елемент об'єкта, що ітерується, `iterable1`.

Функція `map()` може приймати більше одного об'єкта, що ітерується, при необхідності, в цьому випадку наша функція-аргумент повинна приймати таку кількість аргументів, яка відповідає кількості об'єктів, що ітеруються (розглянемо подібний приклад далі).

Давайте виконаємо порівняння реалізацій коду з використанням циклу `for` та із застосуванням функції `map()` для переведення логінів користувачів із списку в нижній регістр.

Перший варіант реалізації на основі циклу `for`:

```
userLogs=['Admin','STUDENT','teacher','User']
userLogsLow=[]

print(userLogs)
```

```
for userLog in userLogs:
    userLogsLow.append(userLog.lower())

print(userLogsLow)
```

Результат:

```
['Admin', 'STUDENT', 'teacher', 'User']
['admin', 'student', 'teacher', 'user']
```

Рисунок 52

Другий варіант реалізації із застосуванням функції `map()`:

```
userLogs=['Admin','STUDENT','teacher','User']
userLogsLow=[]

print(userLogs)
userLogsLow=list(map(str.lower,userLogs))
print(userLogsLow)
```

Забезпечує такий самий результат, але код при цьому більш компактний:

```
['Admin', 'STUDENT', 'teacher', 'User']
['admin', 'student', 'teacher', 'user']
```

Рисунок 53

**Примітка:** зверніть увагу, що як ім'я функції-аргументу ми передаємо `str.lower`, а не просто `lower`, тому що функція-аргумент, що використовується в прикладі, є методом об'єкта `str`.

Розглянемо ще один приклад використання `map()` із вбудованими функціями.

Припустимо, що в нас є список числових значень у форматі рядків (наприклад, отриманий від користувача за допомогою функції `input()`). Для подальшої роботи нам потрібен саме список чисел. Виконаємо перетворення вихідного списку значень у числа за допомогою `map()`:

```
myValues=['2.3','12','45.67','-3']
print(myValues)

myNumbers=list(map(float,myValues))
print(myNumbers)
```

Результат:

```
['2.3', '12', '45.67', '-3']
[2.3, 12.0, 45.67, -3.0]
```

Рисунок 54

`map()` також активно використовується спільно з `lambda`-функціями:

Наприклад, нам потрібно з чисельних значень у форматі рядків одержати першу цифру числа в цілісному форматі.

Реалізація з використанням `lambda` та `map()` буде компактною та зрозумілою:

```
myNumbers=['12.3','12','45.67','30']
print(myNumbers)

myDigits=list(map(lambda x:int(x[0]),myNumbers))
print(myDigits)
```



Результат:

```
['12.3', '12', '45.67', '30']
[1, 1, 4, 3]
```

Рисунок 55

Корисним та зручним є використання `map()` з функціями, визначеними користувачем за допомогою `def`. Наприклад, ми маємо список типів піци, але для формування меню їх потрібно перетворити, додавши до назви типу слово «*Pizza*».

Додавання символу пропуску та слова «*Pizza*» реалізуємо через окрему функцію, яку передамо як аргумент функції `map()` для формування списку назв:

```
pizzaTypes=['Neapolitan','Sicilian','Detroit',
            'New York-Style']
print(pizzaTypes)
def modifyPizzaTypes(pizzaType):
    return pizzaType+' Pizza'

pizzaNames=list(map(modifyPizzaTypes,pizzaTypes))
print(pizzaNames)
```

Результат:

```
['Neapolitan', 'Sicilian', 'Detroit', 'New York-Style']
['Neapolitan Pizza', 'Sicilian Pizza', 'Detroit Pizza',
 'New York-Style Pizza']
```

Рисунок 56

Як було зазначено вище, `map()` може приймати більше одного об'єкта, що ітерується. У такій ситуації ми повинні

попередньо створити таку функцію-аргумент для `map()`, яка приймає кількість аргументів, що відповідає числу ітерованих об'єктів в `map()`.

Розглянемо з прикладу. У нас є список чисел та список степенів, до яких потрібно піднести кожне з чисел першого списку.

```
numbersList1 = [10, 20, 30]
numbersList2 = [1, 2, 3]

result = map(lambda a, b: a**b, numbersList1,
              numbersList2)

print(list(result)) # [10, 400, 27000]
```

За допомогою `map()` ми застосували `lambda`-функцію, яка приймає два аргументи (число та його степінь) для наших двох списків: чисел та їх степенів. Таким чином, забезпечивши, що кількість аргументів `lambda`-функції (два: `a, b`) збіглася з кількістю списків (два: `numbersList1, numbersList2`).

Для завдань фільтрації елементів колекції за певною умовою та їх відбору до нової колекції в Python передбачено функцію `filter()`.

Функція `filter()` приймає два аргументи: функцію, яка реалізує (описує) умову фільтрації і повертає логічні значення (`true` або `false`), і об'єкт, що ітерується, кожен елемент якого буде перевірятися умовою фільтра. Якщо в результаті такої перевірки елемента функція-перевірка поверне `true`, він буде включений в результат роботи фільтра.

Розглянемо на прикладі: ми маємо список із цінами на товари. Припустимо, що потрібно вибрати ціни, значення яких більше ніж 10 доларів.

```
prices=[100.45, 8.56, 5, 234, 45, 87, 567]

expensive=list(filter(lambda x:x>=10, prices))
print(expensive) #[100.45, 234, 45, 87, 567]
```

Тепер потрібно вибрати такі логіни користувачів, які містять слово «user». Перевірку на наявність слова «user» у логіні реалізуємо за допомогою окремої функції `checkLog(user)`, яку передамо функції `filter()`.

```
userLogs=['123user45', 'USERstudent', '56use3', 'user-23',
          'adminUs']

def checkLog(user):
    if user.lower().find('user')!=-1:
        return True
    else:
        return False

selectedUser=list(filter(checkLog, userLogs))

print(selectedUser) # ['123user45', 'USERstudent',
                     'user-23']
```

Також до дуже популярних та корисних функцій відноситься функція `zip()`, яка активно використовується в задачах перебору даних та об'єднання даних із кількох колекцій в один набір.

Наприклад, у нас є два списки: список логінів та паролів користувачів. І необхідно вивести їх у форматі «логін

- пароль», тобто одним циклом пройти по двох списках. Це легко організувати, використовуючи функцію `zip()`:

```
userLogs=['123user45', 'USERstudent', '56use3',
          'user-23', 'adminUs']
userPass=['111', 'abc', '2345', '45fg', 'dffdg']

for log, passw in zip(userLogs, userPass):
    print("login: {} - password: {}".format(log, passw))
```

```
login: 123user45 - password: 111
login: USERstudent - password: abc
login: 56use3 - password: 2345
login: user-23 - password: 45fg
login: adminUs - password: dffdg
```

Рисунок 57

Функція `zip()` приймає колекції, що ітеруються, як аргументи і повертає ітератор. Цей ітератор генерує серію кортежів, які містять елементи кожного об'єкта-аргументу функції `zip()`.

Загальний синтаксис:

```
zip([iterator1, iterator2, .. iteratorN])
```

Функція `zip()` може бути без жодного ітератора, або може містити один, або декілька.

Припустимо, що нам потрібно об'єднати два списки в набір пар елементів:

```
list1=[1,2,3,4,5]
list2=['a','b','c','d','e']
```

```
print(zip(list1,list2))
print(type(zip(list1,list2)))
print(list(zip(list1,list2)))
```

Спочатку подивимося, що повертає функція `zip()` — це ітератор кортежів (об'єкт — `zip object, class 'zip'`). Далі, за допомогою функції `list()`, перетворимо його вміст на список кортежів:

```
<zip object at 0x000001ED896A0DC0>
<class 'zip'>
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e')]
```

Рисунок 58

Розглянемо детальніше отриманий результат: набір кортежів, створюваний `zip()`, формується за правилом: *i*-й кортеж містить *i*-й елемент із кожного списку. Наприклад, перший кортеж `(1, 'a')` складається з перших елементів кожного з двох наших списків.

Таким чином, якщо функції `zip()` передати три списки, вона обробить їх наступним чином:

```
list1=[1,2,3,4,5]
list2=['a','b','c','d','e']
list3=['!','*','#','-','+']

print(list(zip(list1,list2,list3)))
#[(1, 'a', '!'), (2, 'b', '*'), (3, 'c', '#'),
  (4, 'd', '-'), (5, 'e', '+')]
```

При одному об'єкті-аргументі `zip()`, що ітерується, поверне набір кортежів, у кожному з них буде тільки один елемент.

`zip()` можна також використовувати без аргументів. Результатом буде порожній `zip`-об'єкт.

```
list1=[1,2,3,4,5]

print(list(zip(list1)))
#[(1,), (2,), (3,), (4,), (5,)]

print(list(zip())) #[]
```

`zip()` може приймати будь-які типи об'єктів, що ітеруються, такі як файли, списки, кортежі, словники і т. п.

Іноді нам доводиться обробляти нерівні по довжині набори. Ітератор зупиняється, коли найкоротша вхідна колекція вичерпана:

```
list1=[1,2,3]
list2=['a','b','c','d','e']

print(list(zip(list1,list2)))
#[(1, 'a'), (2, 'b'), (3, 'c')]
```

### 3.5. Модуль `functools`

Ми вже знайомі з такими корисними вбудованими функціями вищого порядку як `map()`, `reduce()` і `filter()`. У Python є модуль `functools`, який полегшує та спрощує роботу з функціями вищого порядку. Функція `reduce()`, з якою ми працювали раніше, також входить у `functools`.

Завдяки таким функціям ми можемо повторно застосовувати або розширювати можливості вже створених функцій, не переписуючи їх. Це допомагає нам повторно

використовувати код в інших завданнях досить просто та зручно. Однією з найбільш затребуваних функцій з набору `functools` є функція `partial`, за допомогою якої ми можемо створити нову версію існуючої функції (похідну функцію).

Часткові функції (`partial functions`) — це такі похідні функції, які мають деякі заздалегідь задані вхідні параметри. Наприклад, якщо функція приймає два параметри («`x`» і «`y`»), то з неї можна створити часткову функцію, у якої («`x`» — попередньо заданий аргумент). Далі ми можемо викликати цю часткову функцію лише з одним параметром «`y`».

Давайте розглянемо простий приклад, щоб проілюструвати всі ці нюанси. Припустимо, у нас є функція, яка дублює рядок вказану кількість разів:

```
def myltipleText(myStr, n):
    return myStr*n
```

А якщо нам потрібно і неодноразово реалізовувати повтор рядка два рази, потім ще три рази.

Наприклад, забезпечити такий вивід на основі рядка «*Python*»:

```
PythonPython
PythonPython
PythonPython
PythonPythonPythonPython
PythonPythonPythonPython
PythonPythonPythonPython
PythonPythonPythonPython
```

Можна створити окремі функції, які будуть викликати в собі `myltipleText()` з потрібним параметром (числом повторень):

```
def myltipleText2(myStr):
    return myltipleText(myStr, 2)

def myltipleText3(myStr):
    return myltipleText(myStr, 3)
```

Однак більш зручнішим для таких моментів є використання `partial()`:

```
from functools import partial
def myltipleText(myStr, n):
    return myStr*n

doubleMyltipleText = partial(myltipleText, 2)
tripleMyltipleText = partial(myltipleText, 3)
print(doubleMyltipleText("Python")) #PythonPython
print(tripleMyltipleText("Python")) #PythonPythonPython
```

Як бачимо з нашого прикладу, ми за допомогою `partial()` встановлюємо для аргументу `n` функції `myltipleText()` значення за замовчуванням два (у першому випадку) та три у другому. У цьому прикладі порядок аргументів принципового значення немає. Проте є ситуації, де зміна порядку спричинить некоректний результат роботи функції.

Наприклад:

```
from functools import partial
def SuperCalculation(a,b,c):
    return a**b+c
```



```
result = partial(SuperCalculation,b=2,c=5)
print(result(4)) #21
```

У цьому прикладі порядок дотримання аргументів впливає на правильність обчислень. Тому, щоб зафіксувати змінні-аргументи, ми використовували ключові слова.

Ще одна корисна можливість, що надається модулем `functools` — це кешування, що значно прискорює роботу програми та знижує навантаження на обчислювальні ресурси. Розглянемо засоби `functools`, які дозволять нам кешувати результати наших функцій.

`@lru_cache()` «обертає» нашу функцію з переданими в неї аргументами і «запам'ятовує» результат, який функція повернула після виклику коректно саме з цими аргументами.

Раніше ми вже створювали рекурсивну функцію для обчислення факторіалу числа:

```
def factorialCalculation(n):
    if n==0:
        return 1
    else:
        return n*factorialCalculation(n-1)
```

Щоб кешувати результати виклику функції `factorialCalculation()` ми спочатку імпортуємо `lru_cache` із модуля `functools` та додамо виклик `@lru_cache()` перед функцією `factorialCalculation()`:

```
from functools import lru_cache
@lru_cache
```

```
def factorialCalculation(n):
    if n==0:
        return 1
    else:
        return n*factorialCalculation(n-1)

factorialCalculation(3)
factorialCalculation(6)
factorialCalculation(10)
```

При першому виклику `factorialCalculation(3)` буде виконано чотири запуски функції, але вже при другому виклику `factorialCalculation(6)` інтерпретатор використовує «запам'ятований» результат від `factorialCalculation(3)` і виконає не сім, а лише три, а третій виклик `factorialCalculation(10)` замість десяти запусків виконає лише чотири.

Однак, якщо, у нашої функції `f()` кілька аргументів, заданих за допомогою ключових слів, слід пам'ятати, що виклики `f(a=5, b=10)` і `f(b=10, a=5)` розрізняються по порядку розташування ключових аргументів і можуть мати два окремі записи в кеші.

Кешування — це корисний та ефективний спосіб, за допомогою якого можна прискорити роботу програми та значно знизити навантаження на обчислювальні ресурси.

Використовуючи кешування, ми можемо зберігати останню інформацію або інформацію, що часто використовується в тих місцях пам'яті, які повернуть її за необхідності швидше або дешевше (з погляду обчислювального навантаження), ніж оригінальне джерело інформації.

Наприклад, ми розробляємо програму, яка збирає та відображає останні новини з багатьох різних джерел.

При переході між джерелами новин у списку, наш додаток повинен завантажити з нього новини і показати їх користувачу. Дуже часто користувачі повторно переходять з одного джерела до іншого, і навпаки. Без кешування раніше завантажених даних з джерела новин, нам доведеться, при кожному такому перемиканні, повторно завантажувати з джерела велику порцію одного й того ж контенту. Така ситуація не просто серйозно уповільнить роботу нашої програми, а й спровокує додаткове навантаження на сервер новин, на якому зберігаються статті. Більш правильним та ефективним способом буде зберігати раніше завантажений контент у локальній копії-кеші. І при зверненні користувача (перед завантаженням контенту) перевіряти, чи є така новина в кеші. Якщо є, то завантажувати з кешу, ні — звертатися до сервера новин і виконувати завантаження контенту з нього.

Тепер розглянемо особливості синтаксису та функції `reduce()` — функції «скорочення», яка на відміну розглянутих вище функцій повертає не колекцію, а значення.

Аргументами `reduce()` є деяка функція та послідовність (наприклад, список). Результат — одне обчислене значення, виходячи зі всіх елементів послідовності.

Для кращого розуміння особливостей роботи та переваг використання функції `reduce()`, розглянемо розв'язання задачі знаходження суми елементів списку звичайним способом, використовуючи цикл `for`. Потім її реалізуємо за допомогою функції `reduce()`.

Отже, нам потрібно вирахувати суму елементів списку `[a1, a2, ... an]`:

$$S=a1+a2+....+an.$$

Рішення зводиться до послідовного обчислення проміжних сум:  $S_0=0$  (початкове значення, встановлюється перед обчисленнями)

$$S_1 = S_0 + a_1$$

$$S_2 = S_1 + a_2$$

.....

$$S_n = S_{n-1} + a_n = a_1 + a_2 + \dots + a_n.$$

Обчислення значення  $S_n$  являє собою шукану суму  $S$ . Значення проміжних сум  $S_1, \dots, S_{n-1}$  не потрібно зберігати, тому послідовність обчислень, представлену вище, можна сформулювати у вигляді загальної формули:

$$S = S + a_i,$$

де  $a_i$  — доданок на  $i$ -тому кроці.

Таким чином, обчислення суми зводиться до її накопичення в змінній  $S$  на кожному кроці циклу.

Наприклад, у списку знаходиться послідовність чисел від 1 до 5 включно.

Наші доданки:

$$a_1 = 1$$

$$a_2 = 2$$

$$a_3 = 3$$

$$a_4 = 4$$

$$a_5 = 5$$

$$S = a_1 + a_2 + a_3 + a_4 + a_5$$

$$S = 1 + 2 + 3 + 4 + 5$$

Перед початком обчислення суми ще немає, тобто її значення дорівнює нулю.

$$S_0 = 0$$

$$\begin{aligned}
 S_1 &= S_0 + a_1 = 0 + 1 = 1 \\
 S_2 &= S_1 + a_2 = 1 + 2 = 3 \\
 S_3 &= S_2 + a_3 = 3 + 3 = 6 \\
 S_4 &= S_3 + a_4 = 6 + 4 = 10 \\
 S_5 &= S_4 + a_5 = 10 + 5 = 15
 \end{aligned}$$

Організуємо цикл із накопичення суми:

```

numbers=[1,2,3,4,5]

sumNum=0

for num in numbers:
    sumNum+=num

print("sum is {}".format(sumNum)) #15

```

Ми створюємо змінну `sumNum` і встановлюємо її рівною `0`. Потім ми перебираємо наш список чисел `numbers`, використовуючи цикл `for`, і додаємо кожне число до результату попередньої ітерації (акумулюючи результат). Після проходження по всьому списку чисел, сума (аккумулятор) дорівнюватиме `15`.

Ми можемо виконати розглянуте вище завдання, використовуючи функцію `reduce()` замість циклу `for`.

Синтаксис:

```

reduce (functionName, iterableObj [, initializer])

```

Функція може приймати три аргументи, два з яких є обов'язковими: функція та об'єкт, що ітерується (наприклад, список). Третій аргумент, який є ініціалізатором (початковим значенням), є необов'язковим.

Спочатку функція `functionName` викликається з першими двома елементами послідовності `iterableObj` і повертається результат. Потім функція `functionName` викликається знову з результатом, отриманим на кроці 1, і наступним значенням у послідовності `iterableObj`. Цей процес повторюється доти, доки в послідовності `iterableObj` є елементи. Коли надається третій аргумент (початкове значення, `initializer`), то функція `functionName` викликається з початковим значенням `initializer` і першим елементом з послідовності `iterableObj`.

Наша сума чисел у списку:

$$1 + 2 + 3 + 4 + 5$$

може бути записана так:

$$((((0 + 1) + 2) + 3) + 4) + 5).$$

Нуль тут — те початкове значення (`initializer`), яке не додає до суми нічого, але може слугувати відправною точкою. Також, якщо буде передано порожній список, функція `reduce()` поверне його значення.

Функція `reduce()` знаходиться в модулі `functools`. Отже, щоб її використати, нам потрібно або імпортувати весь модуль `functools`,

```
import functools
```

або ми можемо імпортувати лише функцію `reduce()` із `functools`:

```
from functools import reduce
```

Тепер знайдемо суму наших чисел, використовуючи функцію `reduce()` та створену нами окремо функцію підсумовування двох чисел:

```
import functools
from functools import reduce

numbers=[1,2,3,4,5]

def mySum(x, y):
    return x+y

result1=reduce(mySum, numbers)
print("sum is {}".format(result1)) #15

result2=reduce(mySum, numbers, 0)
print("sum is {}".format(result2)) #15
```

Друге застосування функції `reduce()` пройшло з використанням третього аргументу, початкового значення (рівного нулю, ідея та сама, як ми раніше в циклі встановлювали початкове значення `sumNum=0`).

Якщо ми застосовуємо функцію `reduce()` з порожньою колекцією, не вказавши початкове значення, виникне помилка:

```
numbersEmpty=[]

result3=reduce(mySum, numbersEmpty)
```

А ось при заданому початковому значенні (`initializer`) результат по роботі з порожнім списком поверне сам `initializer` (нуль у нашому прикладі):

```
result4=reduce(mySum, numbersEmpty, 0)
print("sum is {}".format(result4)) #0
```

Ми могли б і не оголошувати функцію `mySum()` окремо, а використовувати `lambda`-функцію просто всередині виклику `reduce()`:

```
numbers=[1,2,3,4,5]
result5=reduce(lambda x,y:x+y, numbers, 0)
print("sum is {}".format(result5)) #15

import functools
from functools import reduce

words=['Python','is','cool']
sentence=reduce(lambda x, y:x+" "+y,words,"")
print(sentence)
```

Розглянемо ще один приклад використання функції `reduce()`. Припустимо, що ми маємо список слів і нам потрібно зібрати їх в речення.

```
import functools
from functools import reduce

words=['Python','is','cool']

sentence=reduce(lambda x, y:x+" "+y,words,"")
print(sentence)
```

Результат:

**Python is cool**

Рисунок 59

Як ми бачимо в результаті, у нас один зайвий пробіл на початку рядка. Змінимо нашу `lambda`-функцію, додавши



до неї перевірку на те, чи є перший елемент порожнім рядком, якщо так, то проводити конкатенацію з наступним словом списку без пробілу.

```
import functools
from functools import reduce

words=['Python', 'is', 'cool']

sentence=reduce(lambda x, y:x+y if x==" " else
                 x+" "+y, words, "")

print(sentence)
```

Результат:

Python is cool

Рисунок 60

### 3.6. Замикання

Ми вже знаємо, що в Python існує чотири області видимості:

- **Локальна область видимості** (*Local*) — це частина коду, яка відповідає тілу деякої функції або лямбда-виразу.
- **Вкладена (нелокальна) область видимості** (*Enclosing*) — така область видимості виникає під час роботи вкладених функцій і містить ідентифікатори, які ми визначаємо у вкладеній функції.
- **Глобальна область видимості** (*Global*) — це рівень програми (скрипту, модуля) Python, який містить усі ідентифікатори, які ми визначили на рівні програми (тобто поза функціями, якщо вони є).

- **Вбудована область видимості (Built-in)** — це рівень мови Python, тобто всі вбудовані об'єкти функції Python знаходяться у цій області.

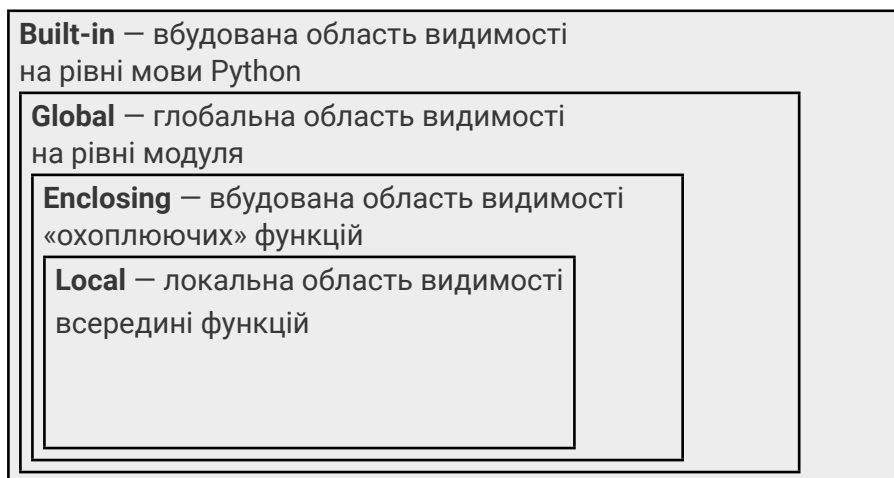


Рисунок 61

Особливості роботи з локальною та глобальною областями видимості ми вже вивчили детально.

Однак поширеним підходом у Python-програмуванні є визначення функції всередині іншої функції. Така функція, визначення якої знаходиться всередині іншої, є вкладеною функцією. Функція, що містить у собі визначення інших функцій, називається охоплюючою функцією.

Ідентифікатори, визначені всередині охоплюючих функцій відносяться до області **Enclosing** і «помітні» з коду вкладених і охоплюючих функцій (тобто локальна змінна охоплюючої функції для її вкладеної функції відноситься до **Enclosing** області видимості). І той момент, коли внутрішня функція посилається на змінні

**Enclosing** області видимості, називається замиканням (*closure*).

За визначенням, замикання — це внутрішня функція, яка посилається на змінні, оголошені в тілі охоплюючої функції. Такі змінні називаються нелокальними (*nonlocal*) змінні.

Розглянемо з прикладу:

```
def sayUserHello(user):
    msg="Hello, " + user

    def showMsf():
        print(msg+"! Let's start...")
```

Для охоплюючої функції `sayUserHello()` вкладена функція `showMsf()` у цьому прикладі є замиканням, а змінна `msg` являється для неї нелокальною (*nonlocal*). Функція `showMsf()` має доступ до неї, щоб працювати з її значенням. Щоб продемонструвати це, додамо в тіло охоплюючої функції `sayUserHello()` виклик функції `showMsf()`:

```
def sayUserHello(user):
    msg="Hello, " + user
    def showMsf():
        print(msg+"! Let's start...")

    showMsf()

sayUserHello('admin') #Hello, admin! Let's start...
```

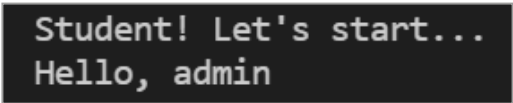
А як щодо можливості змінювати значення нелокальних змінних всередині функції-замикання?

Якщо ми спробуємо це виконати, просто задавши нове значення нелокальної змінної `msg`, то фактично створимо

тільки локальну змінну для функції `showMsf()`, яка теж має ім'я `msg`, використовується далі функцією, але фактично не змінює значення нелокальної версії змінної `msg`. Для демонстрацій цього, ми, після виклику `showMsf()` в тілі `sayUserHello()`, виводимо значення змінної `msg` і бачимо, що воно не було змінено всередині функції `showMsf()`:

```
def sayUserHello(user):  
    msg="Hello, " + user  
  
    def showMsf():  
        msg="Student"  
        print(msg+"! Let's start...")  
  
    showMsf()  
    print(msg)
```

Результат:



```
Student! Let's start...  
Hello, admin
```

Рисунок 62

Для того, щоб забезпечити бажаний результат (надання можливості змінювати значення нелокальних змінних всередині функції-замикання) нам потрібно перед ім'ям такої змінною додати ключове слово `nonlocal`:

```
def sayUserHello(user):  
    msg="Hello, " + user  
  
    def showMsf():  
        nonlocal msg
```

```
msg="Student"
print(msg+"! Let's start...")
showMsf()
print(msg)
```

Результат:

```
Student! Let's start...
Student
```

Рисунок 63

А тепер розглянемо ситуацію, коли наша охоплююча функція не викликає замикання, а повертає його, як результат своєї роботи.

```
def sayUserHello(user):
    msg="Hello, "+ user
    def showMsf():
        print(msg+"! Let's start...")
    return showMsf
```

Оскільки тепер функція `sayUserHello()` повертає результат, то рядок коду із викликом функції `sayUserHello()` також буде змінено: ми створимо змінну, яка прийме результат її роботи — об'єкт функції.

```
case1=sayUserHello('admin')
case1() # Hello, admin! Let's start...
```

Змінна `msg` не просто використовується замиканням, вона «запам'ятовується». Продемонструємо цей нюанс на прикладі: наша функція замикання має власні параметри:

```
def doExercise1(var1):
    var2=5
    def doExercise2(var3):
        return var1**var3
    return doExercise2
```

Функція-замикання `doExercise2()` використовує у своєму тілі лише змінну `var1` (аргумент охоплюючої функції `doExercise1()`). Тому значення змінної `var1` буде «запам'ятоване» (незважаючи на те, що виконання функції `doExercise1(2)` було завершено).

```
case1=doExercise1(2)

print(case1(5)) #32
print(case1(10)) #1024
```

У рядку виклику «`case1(5)`» програма використовує для обчислення «запам'ятоване» значення змінної `var1` (2), а значення `5` використовується як аргумент функції-замикання `doExercise2()`. Також і під час виклику «`case1(10)`» використовується «запам'ятоване» значення змінної `var1` (2) та аргумент для функції-замикання `doExercise2()` — значення `10`.

Розглянемо ще один приклад, який демонструє здатність замикань «запам'ятовувати»: реалізуємо програму — лічильник запусків функції.

```
def launchCounter():
    counter = 0
    def incrementCounter():
        nonlocal cout
```

```

        counter+= 1
        return counter
    return incrementCounter

n=launchCounter()
for i in range(5):
    print(n())

```

Ми запустили наш лічильник у циклі та побачили «запам'ятовування» змінної `counter` після кожного запуску функції.

**Примітка:** *запам'ятовуються лише ті зі змінних, створені в тілі охоплюючої функції, які використовувалися всередині функції-замикання.*

Застосування замикань (крім «запам'ятовування» корисних даних) також дозволяє уникнути використання глобальних змінних.

### 3.7. Каррінг

Тут ми познайомимось зі ще однією популярною технікою функціонального програмування, яка називається каррінг (*currying*).

Каррінг (також вживається термін «карирування») — це техніка перетворення функції від кількох аргументів на послідовність функцій, кожна з яких має лише один аргумент. Тобто, це така трансформація функції  $f(x, y, z)$  для прийняття аргументів у вигляді  $f(x)(y)(z)$ .

Розглянемо з прикладу. Припустимо, що ми маємо функцію, яка приймає два аргументи: логін користувача, якому адресується повідомлення, і сам текст повідомлення:

```
def sendMsg(userTo, msgTxt):
    print("Dear {}, welcome to Python world! {}".
          format(userTo,msgTxt))
```

А це виклики цієї функції в основній програмі:

```
sendMsg('admin', 'Have a nice day!')
sendMsg('admin', 'See you!')
sendMsg('admin', 'Good luck!')
sendMsg('student', 'Good luck!')
```

Як бачимо, у цій точці програми перший аргумент часто повторюється. Було б зручно в цій ситуації мати можливість викликати цю функцію із встановленим (зафіксованим) першим аргументом «**admin**», вказуючи лише другий аргумент — текст повідомлення.

Карирований варіант нашої функції `sendMsg()` виглядатиме так:

```
def sendMsg(userTo):
    def setMsg(msgTxt):
        print("Dear {}, welcome to Python world! {}".
              format(userTo,msgTxt))

    return setMsg
```

Тепер створимо нову функцію для потрібного імені користувача:

```
userAdmin=sendMsg('admin')
```

І викличемо її з потрібними аргументами-повідомленнями:



```

userAdmin('Have a nice day!')

userAdmin('See you!')

userAdmin('Good luck!')

```

Для користувача-студента (якому потрібно надіслати лише одне вітання) будемо безпосередньо викликати функцію `sendMsg()` таким чином (тип користувача — студент є першим параметром функції):

```

sendMsg('student') ('Good luck!')

```

Результат:

```

Dear admin, welcome to Python world! Have a nice day!
Dear admin, welcome to Python world! See you!
Dear admin, welcome to Python world! Good luck!

```

Рисунок 64

Методику карингу можна використовувати з будь-якою кількістю аргументів. Наприклад, нехай наступна версія нашої функції `sendMsg()`, окрім логіна користувача, якому адресується повідомлення, і тексту повідомлення, приймає логін відправника і назву мови програмування, яку почне вивчати користувач (не тільки Python, як у попередньому прикладі).

```

def sendMsg(userTo) :
    def setMsg(msgTxt) :
        def setUserFrom(userFrom) :
            def setLang(lang) :

```

```

        print("Dear {}, Hello from {}.
              Welcome to {} world! {}".
              format(userTo, userFrom,
                    lang, msgTxt))

    return setLang
    return setUserFrom
    return setMsg

```

Зробимо виклики для формування повідомлення для адміністратора та студента від різних відправників та з різними мовами програмування:

```

case1=sendMsg('admin') ('Good luck!')
case2=sendMsg('student') ('See you!') ('admin')

case1('teacher') ('Python')
case2('C++')

```

Результат:

```

Dear admin, Hello from teacher. Welcome to Python world! Good luck!
Dear student, Hello from admin. Welcome to C++ world! See you!

```

Рисунок 65

Як впливає з розглянутих прикладів, карирування дозволяє нам легко створювати частково застосовані функції, які дозволяють спрощувати виклики з недостатнього або частково повторюваного набору аргументів. При таких ситуаціях можна просто передати частину аргументів (які, наприклад, повторюються) у функцію та отримати назад часткову функцію, яка прийматиме інші аргументи.

## 4. Декоратори

Раніше ми вже неодноразово використовували можливості функцій, як об'єктів першого класу: зберігали їх у змінні; передавали, як аргументи, і повертали, як результат роботи іншої функції. Також ми розглянули, наскільки зручною та корисною є методика визначення функції всередині іншої функції. Вивчили переваги роботи з частково застосованими функціями, які дозволяють спрощувати виклики за недостатнього або повторюваного набору аргументів. І зараз ми познайомимось зі ще одним корисним інструментом у Python — декораторами.

Декоратор — це функція-«обгортки», що дозволяє нам розширити функціональність вже існуючої функції без прямої зміни коду в її тілі. Використання декораторів показує, що функція може працювати з іншою функцією, як із звичайними аргументами (даними).

Розглянемо з прикладу.

```
def simpleDecorator(myFunction):  
    print("Hello! I'm Decorator!")  
  
    def simpleWrapper():  
        print("Function starts working...")  
        myFunction()  
        print("See you!")  
    return simpleWrapper
```

У цьому прикладі ми створили декоратор — функцію `simpleDecorator()`, яка (будучи функцією вищого порядку),

як аргумент, приймає ім'я іншої функції `myFunction`, функціональність якої нам потрібно розширити. Всередині `simpleDecorator()` ми визначаємо функцію-«обгортку» `simpleWrapper()`. Функція `simpleWrapper()` «обертає» функцію-аргумент `myFunction`, тобто у своєму тілі містить рядки коду з новою функціональністю та виклик «декорованої» функції `myFunction()`.

Як результат, декоратор повертає функцію-«обгортку». Тепер припустимо, що ми маємо функцію, код якої ми (з різних причин) не можемо змінювати:

```
def sayHi():  
    print("Welcome!")
```

Для зміни (доповнення, розширення) її функціональності ми її «декоруватимемо». Передамо її декоратору `simpleDecorator()`, який за допомогою функції-«обгортки» `simpleWrapper()` додасть нову поведінку і поверне нову версію нашої базової функції `sayHi()` із вже розширеною функціональністю.

```
sayHiAdvanced = simpleDecorator(sayHi)  
sayHiAdvanced()
```

Результат:

```
Hello! I'm Decorator!  
Function starts working...  
Welcome!  
See you!
```

Рисунок 66

Записати код попереднього прикладу, використовуючи синтаксис декораторів за допомогою інструкції «@ім'я\_декоратора» можна так:

```
@simpleDecorator
def sayHi():
    print("Welcome!")
sayHi()
```

Результат буде аналогічний попередньому:

```
Hello! I'm Decorator!
Function starts working...
Welcome!
See you!
```

Рисунок 67

Таким чином, другий спосіб «декорування» функцій — це використання інструкції «@ім'я\_декоратора», яку потрібно помістити над оголошенням функції, що «декорується». Однак, фактично, ми могли просто перевизначити нашу базову функцію `sayHi()` за допомогою першого підходу так:

```
def simpleDecorator(myFunction):
    print("Hello! I'm Decorator!")
    def simpleWrapper():
        print("Function starts working...")
        myFunction()
        print("See you!")
    return simpleWrapper
def sayHi():
```

```
print("Welcome!")
sayHi = simpleDecorator(sayHi)
sayHi()
```

Результат:

```
Hello! I'm Decorator!
Function starts working...
Welcome!
See you!
```

Рисунок 68

Ми можемо використовувати один декоратор для будь-якої функції та декорувати функцію будь-яким (або декількома) декоратором.

Створимо ще одну базову функцію:

```
def sayBye():
    print("Buy!")
```

І «декоруємо» її створеним раніше декоратором `simpleDecorator`:

```
sayBye = simpleDecorator(sayBye)
sayBye()
```

Результат:

```
Hello! I'm Decorator!
Function1 starts working...
Buy!
See you!
```

Рисунок 69

Тепер «декоруємо» нашу першу базову функцію `sayHi()` двома різними декораторами. Для цього спочатку створимо другий декоратор:

```
def simpleDecorator_v2(myFunction):
    print("Hello! I'm Second Decorator!")
    def simpleWrapper():
        print("Let's start...")
        myFunction()
        print("Good luck!")
    return simpleWrapper
```

І застосуємо обидва до функції `sayHi()`:

```
@simpleDecorator
@simpleDecorator_v2
def sayHi():
    print("Welcome!")
sayHi()
```

Результат:

```
Hello! I'm Second Decorator!
Hello! I'm Decorator!
Function starts working...
Let's start...
Welcome!
Good luck!
See you!
```

Рисунок 70

У випадку, коли до функції застосовується кілька декораторів, вони спрацювуватимуть у порядку, зворотному до того, в якому вони були викликані.

Тобто виклик:

```
@simpleDecorator
@simpleDecorator_v2
.....
```

можна для кращого розуміння записати так:

```
sayHi = simpleDecorator(simpleDecorator_v2(sayHi))
sayHi()
```

Поки що ми розглянули лише ситуації, коли базові функції не повертали значень. Якщо ж функція, яку потрібно «декорувати», має повертати значення, то його також має повертати і функція-«обгортка».

Наприклад:

```
def simpleDecorator_v3(myFunction):
    print("Hello! I'm Third Decorator!")
    def simpleWrapper():
        print("Function starts working...")
        resutl=myFunction()
        print("See you!")
        return resutl
    return simpleWrapper

def calculateSum():
    print("Welcome! Let's calculate...")
    x=int(input("x: "))
    y=int(input("y: "))
    return x+y

calculateSum = simpleDecorator_v3(calculateSum)
print(calculateSum())
```



```

Hello! I'm Third Decorator!
Function starts working...
Welcome! Let's calculate...
x: 1
y: 2
See you!
3

```

Рисунок 71

Тож ми вже навчилися створювати декоратори для функцій, які повертають результат. Але в усіх попередніх прикладах базові функції не мали аргументів. Тому зараз потрібно розглянути, яким чином можна передати аргументи функції, що декорується.

Аналогічно до попередньої ситуації з поверненням, результати функція-«обгортка» і в цьому випадку має забезпечити передачу аргументів у функцію, що декорується:

```

def simpleDecorator_v4(myFunction):
    print("Hello! I'm Fourth Decorator!")
    def simpleWrapper(argX, argY):
        print("I've got {},{}.Function starts working...".
              format(argX, argY))
        resutl=myFunction(argX, argY)
        print("See you!")
        return resutl
    return simpleWrapper

def calculateSum_v1(a,b):
    print("Welcome! Let's calculate...")
    x=int(input("x: "))
    y=int(input("y: "))
    return x+y+a+b

```

```
calculateSum_v1 = simpleDecorator_v4(calculateSum_v1)
print(calculateSum_v1(3,4))
```

Результат:

```
Hello! I'm Fourth Decorator!
I've got 3, 4. Function starts working...
Welcome! Let's calculate...
x: 1
y: 2
See you!
10
```

Рисунок 72

Якщо можна передавати аргументи функції, що декорується, то виникає питання: а чи можна передавати аргументи самому декоратору? Адже ми знаємо, як аргумент декоратор має приймати базову функцію. А якщо нам потрібні якісь додаткові дані, наприклад, для управління логікою роботи самого декоратора?

Для вирішення цього завдання нам потрібно додати ще один рівень абстракції — створити «обгортку» для самого декоратора і передати цій функції потрібні додаткові аргументи.

Обов'язкова умова: ця функція-«обгортка» для декоратора має повертати декоратор в результаті своєї роботи.

```
def decoratorWrapper(argForDec) :
    print("I've got arg={} for decorator!".
          format(argForDec))
```

```
def simpleDecorator_v5(myFunction):
    print("Hello! I'm Decorator with arg={}!".
          format(argForDec))

    def simpleWrapper(argX, argY):
        print("Hi! I am Function. I've got {}, {}".
              format(argX, argY))
        result=myFunction(argX, argY)+argForDec

        print("See you!")
        return result

    return simpleWrapper
```

Тепер ми викличемо нашу функцію `decoratorWrapper()` з потрібним аргументом та отримаємо декоратор, якому вони були передані:

```
decoratorWithArg =decoratorWrapper(10)
```

Далі «декоруємо» нашу базову функцію з двома аргументами `calculateSum_v1()` та викличемо її нову «декоровану» версію:

```
def calculateSum_v1(a,b):
    print("Welcome! Let's calculate...")
    x=int(input("x: "))
    y=int(input("y: "))

    return x+y+a+b

calculateSum_v1 = decoratorWithArg(calculateSum_v1)
print(calculateSum_v1(3,4))
```

Результат:

```
I've got arg=10 for decorator!
Hello! I'm Decorator with arg=10!
Hi! I am Funcion. I've got 3, 4. Function starts working...
Welcome! Let's calculate...
x: 1
y: 2
See you!
20
```

Рисунок 73

Розглянемо приклад використання декоратора. Припустимо, що ми маємо список із цінами на товари в доларах. І функція, яка переводить ціну товару в доларах у відповідний гривневий еквівалент:

```
pricesUSD=[100.34,35,67.99,25.5]
print(pricesUSD)

def toPriceNew(priceList):
    return list(map(lambda x: x*27.5, priceList))
```

Однак зараз на всі товари діє знижка (наприклад, 15%) і нам потрібно перевести ціни в гривні, та ще додатково врахувати знижку.

Знижка — непостійна характеристика товару, тому змінювати код функції `toPriceNew()` немає сенсу.

Ми створимо декоратор, який «застосує» знижку після переведення ціни в іншу валюту:

```
pricesUSD=[100.34,35,67.99,25.5]
print(pricesUSD)
def toPriceNew(priceList):
```

```

    return list(map(lambda x: x*27.5, priceList))
def changePriceDecorator_v1(myFunction):
    print("Hello! Let's change your prices...")
    def simpleWrapper(argList):
        print("I've got list of prices with {} elements.
              Function starts working...".
              format(len(argList)))
        resutl=myFunction(argList)
        resutlwithDisc=list(map(lambda x: x*(1-0.15),
                                resutl))
        print("Let's set a discount..")
        return resutlwithDisc
    return simpleWrapper

pricesToGRN = changePriceDecorator_v1(toPriceNew)
print(pricesToGRN(pricesUSD))

```

Результат:

```

[100.34, 35, 67.99, 25.5]
Hello! Let's change your prices...
I've got list of prices with 4 elements. Function starts working...
Let's set a discount..
[2345.4474999999998, 818.125, 1589.26625, 596.0625]

```

Рисунок 74

Однак наша знижка не завжди може становити саме 15%. Зручніше розмір знижки передати декоратору, як його аргумент.

Змінимо попередній код наступним чином:

```

pricesUSD=[100.34, 35, 67.99, 25.5]
print(pricesUSD)

def toPriceNew(priceList):

```

```

    return list(map(lambda x: x*27.5, priceList))

def setDiscountDecoratorWrapper(disc):
    def changePriceDecorator_v1(myFunction):
        print("Hello! Let's change your prices...")
        def simpleWrapper(argList):
            print("I've got list of prices with {}
                  elements. Function starts working...".
                  format(len(argList)))
            resutl=myFunction(argList)
            resutlwithDisc=list(map(lambda x: x*(1-disc),
                                     resutl))
            print("Let's set a discount..")
            return resutlwithDisc
        return simpleWrapper
    return changePriceDecorator_v1

discount=float(input("Discount value:"))
changePriceDecorator_v2=
    setDiscountDecoratorWrapper(discount)

pricesToGRN = changePriceDecorator_v2(toPriceNew)
print(pricesToGRN(pricesUSD))

```

Результат:

```

[100.34, 35, 67.99, 25.5]
Discount value:0.25
Hello! Let's change your prices...
I've got list of prices with 4 elements. Function starts
working... Let's set a discount..
[2069.5125, 721.875, 1402.2937499999998, 525.9375]

```

Рисунок 75

Використовуючи такий підхід, ми можемо передавати декораторам потрібні їм аргументи у будь-якій кількості.





## Урок 4

### Функції

© STEP IT Academy, [www.itstep.org](http://www.itstep.org)

© Анна Егошина

Усі права на фото-, аудіо- і відеотвори, що охороняються авторським правом і фрагменти яких використані в матеріалі, належать їх законним власникам. Фрагменти творів використовуються в ілюстративних цілях в обсязі, виправданому поставленим завданням, у рамках учбового процесу і в учбових цілях, відповідно до законодавства про вільне використання твору без згоди його автора (або іншої особи, яка має авторське право на цей твір). Обсяг і спосіб цитованих творів відповідає прийнятим нормам, не завдає збитку нормальному використанню об'єктів авторського права і не обмежує законні інтереси автора і правовласників. Цитовані фрагменти творів на момент використання не можуть бути замінені альтернативними аналогами, що не охороняються авторським правом, і відповідають критеріям добросовісного використання і чесного використання.

Усі права захищені. Повне або часткове копіювання матеріалів заборонене. Узгодження використання творів або їх фрагментів здійснюється з авторами і правовласниками. Погоджене використання матеріалів можливе тільки якщо вказано джерело.

Відповідальність за несанкціоноване копіювання і комерційне використання матеріалів визначається чинним законодавством.