**Лекція 2**

**Тема . Елементи функціонального програмування**

План

1. Функція: визначення і виклик.
2. Області видимості в мові Python.
3. Анонімні функції: lambda.
4. Рекурсія.
5. Функції range () та xrange ().
6. Функції map, filter, reduce, zip.
7. **Функція: визначення і виклик.**

**Функції** - це основні програмні структури в мові Python, що забезпечують багаторазове використання програмного коду і зменшують його надлишковість.

Функції - це ще й засіб проектування, який дозволяє розбити складну систему на досить прості і легко керовані частини. У процесі розробки функції відіграють дві основні ролі: максимізувати багаторазове використання програмного коду і мінімізувати його надлишковість.

Python надає у використання вбудовані функції, на зразок, **len()**, **print()**, **input()** тощо. Але можна створювати і власні функції.

Функція може приймати будь-яку кількість будь-яких вхідних параметрів і повертати результат у вигляді будь-яких об’єктів (списків, кортежів тощо).

З функцією можна зробити дві речі:  
–визначити;  
– викликати.

Для **визначення** функції, необхідно написати **def**, ім’я функції, вхідні параметри у круглих дужках, двокрапку, після якої з відступом йде блок коду:

def назва\_функції(вхідні параметри):

блок коду

def birthday\_card():

print('Happy birthday!')

birthday\_card()

Отримаємо: 'Happy birthday!'

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

***Параметри функції***

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

***Формальні параметри*** – параметри, що вказуються при  
оголошенні функції.

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

def print\_numbers(limit):

for i in range (limit):

print(i)

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

print\_numbers(n)

Введіть кількість елементів: 5

0

1

2

3

4

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

При необхідності повернути результат роботи функції в програму, з якої вона викликалася, для її подальшого оброблення застосовується команда ***return***. Вираз, що стоїть після *return* буде повертатися в якості результату виклику функції. Без аргументів *return* використовується для виходу з функції (інакше вихід відбудеться при досягненні кінця функції).

В Python функції здатні повертати кілька значень одночасно.

def PrintRoots(a, b, c):

D = b\*\*2 - 4 \* a \* c

import math

x1 = (-b + math.sqrt(D)) / 2 \* a

x2 = (-b - math.sqrt(D)) / 2 \* a

return x1, x2

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

Результатом запуску даного коду буде:

(1.0, -1.0)

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

x1, x2 = PrintRoots(1.0, 0, -1.0)

print("x1 =", x1, "\nx2 =", x2)

Результатом запуску даного коду буде:

x1 = 1.0

x2 = -1.0

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

def traffic\_light (color):

if color == 'red':

return "STOP!"

elif color == "green":

return "GO!"

elif color == 'yellow':

return "GET READY!"

else:

return "Broken traffic light!"

Викликавши функцію *traffic\_light()*, передавши їй в якості аргументу рядок 'blue'.

result = traffic\_light('blue')

print(result)

Результатом буде: 'Broken traffic light!'

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

def do\_nothing():

pass

Результатом буде: None

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

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

def summa (x, y):

return x + y

print(summa(5,6))

f = summa

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

print(v)

Результатом запуску даного коду буде:

11

1 3

***Аргументи функцій***

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

***Позиційні аргументи***

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

def func(a, b, c):

return a+b\*c

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

Отримаємо:

7

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

***Іменовані аргументи***

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

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

Отримаємо:

5

Можна об’єднувати позиційні аргументи та іменовані аргументи.

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

Отримаємо:

8

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

***Значення параметра за замовчуванням***

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

def func(a, b, c=2):

return a+b\*c

Викликаючи функцію func() можна не передавати їй аргумент с:

print(func(1, 2))

Отримаємо:

5

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

print(func(1, 2, 3))

Отримаємо:

7

***Функція зі змінним числом аргументів***

Якщо перед параметром у визначенні функції вказати **символ \***, то функції можна буде передати будь-яку кількість параметрів. Всі передані параметри зберігаються в кортежі. У наступному прикладі args є кортежем параметрів, який був створений з аргументів, переданих у функцію print\_args():

def print\_args(\*args):

# функція приймає будь-яку кількість параметрів

print('Кортеж позиційних аргументів:', args)

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

def variable\_len(\*args):

for x in args:

print(x)

variable\_len(1,2,3) # Виведе 1,2,3

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

def print\_more(numb\_1, numb\_2, \*args):

print(numb\_1)

print(numb\_2)

print(args)

print\_more(1, 2, 3, 4, 5, 6)

Результатом запуску даного коду буде:

1

2

(3, 4, 5, 6)

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

Замість однієї зірочки можна використовувати **дві \*\*,** щоб згрупувати іменовані аргументи в словник, де імена аргументів стануть ключами, а їх значення – відповідними значеннями в словнику. У наступному прикладі визначається функція *print\_kwargs ()*, в якій виводяться її іменовані аргументи:

def print\_kwargs(\*\*kwargs):

print('Іменовані аргументи:', kwargs)

Викликавши її, передавши кілька аргументів:

print\_kwargs(a = 1, b = 2, c = 3)

Отримаємо наступний результат:

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

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

def variable\_len(\*\*args):

print(type(args))

for x, value in args.items():

print(x, value)

variable\_len(apple = "яблуко", bread = "хліб")

# Виведе apple яблуко bread хліб

***Потік виконання***

З появою функцій програми перестали бути лінійними, в зв’язку з цим виникло поняття потоку виконання – послідовності виконання інструкцій, що складають програму. Виконання програми, написаної на Python завжди починається з першого виразу, а наступні вирази виконуються один за іншим зверху вниз.

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

З тіла будь-якої функції може бути викликана інша функція, яка теж може в своєму тілі містити виклики функцій і т.д. Проте, інтерпретатор Python пам’ятає звідки була викликана кожна функція, і рано чи пізно, якщо під час виконання не виникне ніяких винятків, він повернеться до вихідного виклику, щоб перейти до наступної інструкції.

def func1(name):

print('Привіт, '+name)

def func2():

return input('Введіть ім\'я ')

func1(func2())

Результатом запуску даного коду буде:

Введіть ім'я Alex

Привіт, Alex

***Внутрішні функції***

Можна визначити функцію всередині іншої функції:

def outer(a, b):

def inner(c, d):

return c + d

return inner(a, b)

print(outer(4, 7))

Результатом запуску даного коду буде:

11

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

1. **Області видимості в мові Python.**

Область видимості змінних в мові програмування Python являє собою якийсь простір імен, в рамках якого функціонують створені об'єкти. Ця особливість дозволяє обмежувати доступ до певних значень, щоб уникнути конфліктів між однаковими ідентифікаторами. Змінні бувають двох видів: локальні і глобальні, що в більшості випадків визначається місцем їх первинної ідентифікації в програмі.

***Локальні змінні***

Для створення змінних, що володіють локальної областю видимості, необхідно всього лише помістити їх в окремий блок коду, ізольований від решти програми. Щоб побачити локальну змінну в дії, досить форматувати цілочисельний об'єкт з ім'ям *x* і значенням 100 в функції *f*, як це зроблено в наступному прикладі:

def f():

x = 100

print(x)

f()

100

Тут *x* має локальну область видимості, так як доступна лише в рамках своєї функції *f*. Викликаючи цю функцію із зовнішньої частини програми, можна побачити висновок цілочисельного значення на екрані. Однак, якщо спробувати вивести змінну x за допомогою методу *print* поза зоною дії функції *f*, компілятор тут же видасть помилку:

def f():

x = 100

print(x)

f()

print(x)

100

Traceback (most recent call last):

File "main.py", line 5, in <module>

print(x)

NameError: name 'x' is not defined

Так відбувається через те, що ***зовнішня частина програми нічого не знає про змінну x, оскільки містить в собі зовсім інший простір імен***. Користуватися локальними об'єктами можна тільки в тій області, де вони були ідентифіковані. У зворотному ж випадку компілятор повідомить про помилку, не зумівши виявити необхідну змінну.

***Глобальні змінні***

Щоб мати можливість використовувати деяке значення в будь-якій частині програми, слід оголосити глобальну змінну. Для цього знадобитися створити змінну окремо від області коду, обмеженою певним блоком коду, наприклад, функцією. У наступному прикладі демонструється ідентифікація целочисленного типу даних під назвою x, який пізніше виводиться на екран за допомогою методу print в функції ***f***:

x = 100

def f():

print(x)

f()

print(x)

100

100

З викладеного вище результатів виконання програми, значення 100 відтворюється не тільки через ***f***, а й за допомогою звичайного ***print***. Таким чином, отримання доступу до ***x*** здійснюється з будь-якої частини коду, завдяки глобальному контексті подібного об'єкта. Але що буде, якщо спробувати змінити значення глобальної змінної в деякій функції? Результати такого експерименту представлені в наступному фрагменті коду:

x = 100

def f():

x = 200

f()

print(x)

100

Функція ***f*** привласнює значення 200 змінної з ім'ям ***x***, однак, всупереч очікуванням, зовнішній метод print виводить число 100, яке належало ***x*** спочатку. Відбувається так тому, що в даній програмі створюються два різних об'єкта ***x*** з локальної, а також глобальної областю видимості. Виправити ситуацію допоможе ключове слово ***global***:

x = 100

def f():

global x

x = 200

f()

print(x)

200

***Помітивши змінну x як global, можна звертатися до її початкового значення, яке було визначено поза зоною дії функції f***. Тепер після того як в ***x*** помістили число 200, виклик методу ***print*** виводить цілком очікуваний результат, тобто змінене значення.

В Python глобальні змінні можна використовувати для зберігання якихось то налаштувань програми або розроблюваного модуля. Для цього добре підійдуть словники.

Але все таки не варто зловживати. Найчастіше набагато правильніше передавати у функції необхідні значення в якості аргуменов, а якщо потрібно перезаписати якесь глобальне значення, то повертати його з функції.

def my\_abs (val):

     return -val if val <0 else val

     return val + 1

x = -15

x = my\_abs (x)

***Нелокальні змінні***

Отже, для звернення до глобальної змінної всередині функції ***f*** необхідно використовувати ключове слово ***global*** перед її ідентифікатором. Але що якщо потрібно викликати зовсім невеликої, а змінну, яка була визначена в зовнішньому методі, будучи при цьому локальної для іншого простору імен, що знаходиться на рівень вище? Наступний код демонструє спробу взаємодії зі значенням із зовнішнього ***функції f1*** в ***методі f2***:

def f1 ():

     x = 100

     def f2 ():

         x = 200

     f2 ()

     print (x)

f1 ()

100

Незважаючи на те, що змінної з таким же ім'ям x було присвоєно нове значення 200, в результаті виконання написаних методів на екрані відобразилося 100. Як і в тому випадку з двома різними змінними, локальної та глобальної, тут є також два різних об'єкта, які ідентифіковані в окремих блоках коду. Щоб звернутися до об'єкта, який не є локальним, необхідно скористатися модифікатором ***nonlocal***:

def f1():

x = 100

def f2():

nonlocal x

x = 200

f2()

print(x)

f1()

200

Таким чином, в ***методі f2*** здійснюється запис значення 200 в змінну ***x*** з ***функції f1***. В результаті подібних дій, виклик методу ***f1*** з зовнішньої частини програми створює нову змінну x, значення якої змінюється в ***f2*** зі 100 на 200 і виводиться за допомогою ***print***.

***Важливо зауважити, що конструкція nonlocal була додана тільки в 3-ій версії мови Python***.

Видимість із завантажуваного модуля

Тепер розберемося з видимістю глобальних змінних між завантажуваними модулями Python. Наприклад, ми підключаємо інший модуль за допомогою команди import. Створимо файл "test.py" і в нього запишемо наступний код:

print ('Завантажується модуль test')

x = 100

def f ():

     print ('З функції x =' + str (x))

Тобто ми визначили глобальну змінну ***x*** для модуля ***test***. Так само визначили функцію, яка виводить на екран її значення.

Тепер створимо файл main.py, який і будемо запускати. У ньому ми імпортуємо модуль ***test***, а так само створимо свою глобальну змінну x. Після цього виведемо значення глобальної змінної з ***test***, виклик функції ***f***, а також перевіримо, що значення змінної в модулі ***main*** не змінилося:

x = 200

print ('З модуля main x =' + str (x))

import test

print ('З модуля test x =' + str (test.x))

print ('Надаємо значення 300')

test.x = 300;

print ('З модуля test x =' + str (test.x))

test.f ()

print ('З модуля main x =' + str (x))

З модуля main x = 200

Завантажується модуль test

З модуля test x = 100

Надаємо значення 300

З модуля test x = 300

З функції x = 300

З модуля main x = 200

Ми в першому рядку записали в ***x*** значення 200. Це було зроблено, щоб показати, що після того, як ми завантажимо зовнішній модуль, значення цієї змінної не зміниться. Так і вийшло. Звертаючись до змінної з завантаженої бібліотеки, вдалося прочитати його і змінити значення.

Тепер модифікуємо програму в такий спосіб:

x = 200

print ('З модуля main x =' + str (x))

from test import \*

f ()

print ('З модуля main x =' + str (x))

print ('Надаємо значення 300')

x = 300

f ()

print ('З модуля main x =' + str (x))

З модуля main x = 200

Завантажується модуль test

З функції x = 100

З модуля main x = 100

Надаємо значення 300

З функції x = 100

З модуля main x = 300

В цьому випадку для завантаження ми використовували команду "from test import \*". Ми імпортували всі змінні і функції. Після завантаження модуля значення змінної ***x*** в модулі ***main*** змінилося. Але при виконанні функції, ми отримуємо значення ***x*** з модуля ***test***. Після присвоєння нового значення змінної ***x***, значення, яке виводить функція ***f*** не змінюється.

Слід по можливості уникати підключення бібліотек за допомогою команди **from бібліотека import \***, а підключати тільки необхідні функції - **from бібліотека import функція**. При цьому треба впевнитися, що ці імена не використовуються в основному модулі.

Використання **import бібліотека** допоможе уникнути можливих помилок, якщо в програмі є функції, класи і змінні з такими ж назвами, як і у переданому модулі.

***Глобальні змінні в класі***

Точно так само як і в функціях, можна звертатися до глобальних змінних і в класі Python. Розберемо приклад:

x = 100

print(x)

class c1:

global x

x = 200

def \_\_init\_\_(self):

global x

x = 300

def f(self):

global x

x = 400

print(x)

o1 = c1()

print(x)

o1.f()

print(x)

100

200

300

400

Ми оголосили глобальну змінну ***x***. Вивели значення змінної до і після оголошення класу. Як бачимо значення змінилося. Після того як ми створили об'єкт класу, значення в черговий раз змінилося. Це сталося, тому що спрацював конструктор класу - метод \_\_init\_\_. Після виклику функції ***f*** у створеного об'єкта, значення стало 400. В Python використання ***global*** змінна і в функції класу, і в його конструкторі, і після опису класу дають можливість зміни глобальної змінної. Якщо прибрати це оголошення, то тоді виконається привласнення локальної змінної.

Звичайно ж якщо ми визначимо локальну змінну в класі, то до неї не буде можливості доступу з іншого класу:

class c1:

     x = 100

class c2:

     def f (self):

         print (x)

o = c2 ()

o.f ()

Traceback (most recent call last):

   File "main.py", line 7, in <module>

     o.f ()

   File "main.py", line 5, in f

     print (x)

NameError: name 'x' is not defined

Для того, щоб код працював, змінна ***x*** повинна бути глобальною.

Таким чином, область видимості змінних в мові програмування Python, є важливою складовою платформи. Правильна взаємодія з усіма її особливостями дозволяє уникнути безлічі досить складних помилок. Для більш безпечного контролю над видимістю окремих об'єктів застосовуються ключові слова ***global*** та ***nonlocal***.

1. **Анонімні функції: lambda.**

Python дозволяє в короткій формі оголошувати невеликі анонімні функції - lambda -функції.

У загальному вигляді lambda-вираз складається з ключового слова lambda, за яким слідують один або більше аргументів (точно так само, як список аргументів у круглих дужках в заголовку інструкції def) і далі, слідом за двокрапкою, знаходиться вираз:

***lambda argument1, argument2, ... argumentN: вираз***, ***що використовує аргументи***

Як результат lambda-вирази повертають точно такі ж об'єкти функцій, які створюються інструкціями def, але тут є кілька відмінностей, які роблять lambda-вирази зручними в деяких спеціалізованих випадках:

• lambda - це вираз, а не інструкція. З цієї причини ключове слово lambda може з'являтися там, де синтаксис мови Python не дозволяє використовувати інструкцію def, - всередині літералів або у викликах функцій, наприклад. Крім того, lambda-вираз повертає значення (нову функцію), яке при бажанні можна привласнити змінній, на противагу інструкції def, яка завжди пов'язує функцію з ім'ям в заголовку, а не повертає її у вигляді результату.

• Тіло lambda - це не блок інструкцій, а єдиний вираз. Тіло lambda -виразу рівно тому, що ви ставите в інструкцію return всередині визначення def, - ви просто вводите результат у вигляді виразу замість його явного повернення. Внаслідок цього обмеження lambda-виразу менш універсальні, ніж інструкція def - в тілі lambda-виразу може бути реалізована тільки логіка, що не використовує такі інструкції, як if. Така реалізація передбачена заздалегідь - вона обмежує можливість створення великого числа рівнів вкладеності програм: lambda-вирази призначені для створення простих функцій, а інструкції def - для вирішення більш складних завдань. Вони ведуть себе точно так само, як і звичайні функції, які оголошуються ключовим словом def.

Анонімні функції можуть містити лише один вираз, але й виконуються вони швидше.

Анонімні функції створюються за допомогою інструкції lambda:

add = lambda x, y: x + y

Та ж сама функція add може бути визначена за допомогою ключового слова def:

def add(x, y):

return x + y

Виконавши код

print(add(7, 8))

ми отримаємо однаковий результат в обох випадках:

15

Крім цього, анонімну функцію не обов’язково присвоювати змінній

print((lambda x, y: x + y)(5, 12))

і lambda-функції, на відміну від звичайної, не потрібна інструкція return:

17

При виконанні лямбда-функції обчислюється її вираз, а потім результат виразу автоматично повертається, тому завжди існує неявна інструкція return.

>>> def func(x, y, z): return x + y + z

...

>>> func(2, 3, 4)

9

Та такого ж ефекту можна досягти за допомогою lambda-виразу,

>>> f = lambda x, y, z: x + y + z

>>> f(2, 3, 4)

9

У lambda-виразах точно так же можна використовувати аргументи зі значеннями за замовчуванням:

>>> x = (lambda a = "fee", b = "fie", c = "foe": a + b + c)

>>> x ("wee")

'Weefiefoe'

В яких випадках в програмному коді слід застосовувати lambda-функції? Дуже часто, як вище було наголошено, лямбда-функції використовують для написання коротких функцій для сортування об’єктів за альтернативним ключем:

months = [(1, 'January'), (9, 'September'), (7, 'July'), (4, 'March')]

print(sorted(months, key=lambda x: x[1]))

print(sorted(months))

У наведеному вище прикладі ми сортуємо список кортежів по другому значенню в кожному кортежі. В даному випадку лямбда-функція забезпечує швидкий спосіб зміни порядку сортування:

[(1, 'January'), (7, 'July'), (4, 'March'), (9, 'September')]

[(1, 'January'), (4, 'March'), (7, 'July'), (9, 'September')]

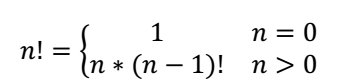
Лямбда найбільш корисні у випадках, коли потрібно визначити багато дрібних функцій і запам’ятати усі їх імена.

1. **Рекурсія.**

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

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

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



def factorial (n):

if n>0:

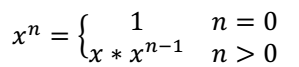
return n\* factorial(n-1)

else:

return 1

print(factorial (5))

Отримаємо: 120



x=10

def rec\_func (n):

if n>0:

return x\* rec\_func (n-1)

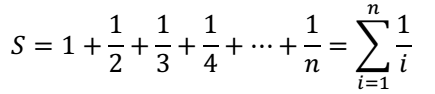
else:

return 1

print(rec\_func (5))

Отримаємо: 100000

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



def rec\_func\_2(n, i):

if i==n:

return 1/n

else:

return 1/i+rec\_func\_2(n, i+1)

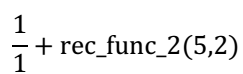
print(rec\_func\_2(5, 1))

Отримаємо: 2.283333333333333

Можна простежити, як працює функція rec\_func\_2, наприклад, для n = 5.

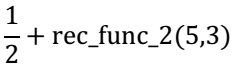
rec\_func\_2(5, 1);

При виконанні тіла функції сформується наступне:

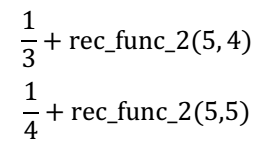


Що знову змушує звернутися до функції rec\_func\_2(5, 2), що

призводить до появи нового значення:



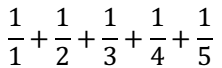
Після виконання ще двох звернень ситуація виявиться наступною:



Потім при черговому виклику функції rec\_func\_2(5, 5) рекурсивні звернення припиняться і буде повернено значення .

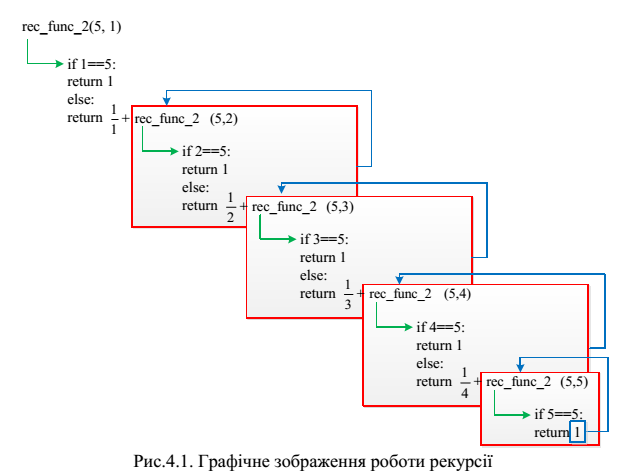
В результаті сформується така послідовність: значення буде передано до + rec\_func\_2(5,5) замість rec\_func\_2(5,5), потім до + rec\_func\_2(5, 4) замість rec\_func\_2(5, 4) і т.д.

В результаті отримаємо ряд:



Ця послідовність операторів і дає результат обчислення суми:

Сума =2.28333333





1. **Функції range () та xrange ().**

Тип range (діапазон) є вбудованої різновидом даних в мові Python, який призначений для зберігання інформації про арифметичній прогресії цілих чисел. Для її створення використовується однойменна функція range з трьома різними параметрами. Діапазон забезпечує комфортну роботу з колекціями при їх циклічній обробці.

***Що таке діапазон?***

Діапазон являє собою незмінну послідовність цілих чисел, яка найчастіше застосовується для генерації більш складних наборів даних за допомогою генераторів. Але на відміну від списків, кортежів, а також інших стандартних колекцій, для обробки такого об'єкта завжди потрібно однаково мала кількість пам'яті. Завдяки особливому підходу до зберігання даних, в пам'ять записуються в повному обсязі елементи послідовності, а тільки її довжина, крок і стартова точка. Інші дані обчислюються в міру необхідності.

***Функція range***

Для генерації діапазону потрібно викликати функцію range, передавши їй від 1 до 3 цілочисельних аргументів. У мові Python діапазон є самостійним об'єктом. Тому створити його можна за допомогою присвоєння результату роботи range визначеному ідентифікатору. Перевірити, до якого класу належить такий об'єкт, можна за допомогою методу type ().

print (type (range (10)))

<Class 'range'>

Розглянемо опис функції range в Python 3. Як вже було сказано, вона приймає кілька параметрів:

1. Початок діапазону (start);
2. Кінець діапазону (stop);
3. Крок діапазону (step).

Все вказувати необов'язково, так як start і step за замовчуванням мають значення 0 і 1 відповідно. Однак самостійно задати stop для діапазону все ж необхідно.

Якщо задати тільки один аргумент, то задається кінець діапазону (stop). Якщо задається аргументів більше одного, то порядок буде наступним: start, stop, step.

Наступний приклад демонструє результат роботи функції range.

data = range(10)

print(data)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Таким чином, вийшла послідовність з 10 цілими елементами, так як значення параметрів start і step залишалися за замовчуванням. Спробуємо додати один аргумент.

data = range(3, 10)

print(data)

[3, 4, 5, 6, 7, 8, 9]

Як видно з результатів виконання програми, перший параметр функції range задає стартове значення для майбутньої послідовності (3), в той час як другий відображає її кінцеву межу (10). Однак, всупереч очікуванням, число 10 не включається в підсумкову послідовність. Додамо третій аргумент у функцію і подивимося, що буде.

data = range(3, 10, 2)

print(data)

[3, 5, 7, 9]

В даному випадку задається діапазон чисел в Python3 використовуючи максимально можливу кількість параметрів, які відповідають за початок (3), кінець (10) і крок (2). Таким чином, функція print вивела 4 цілочисельних значення з однаковим інтервалом послідовності. Варто завжди враховувати порядок розташування її аргументів: start, stop і step. Всі вони можуть володіти лише цілочисельним значенням, як позитивним, так і від'ємним.

Дуже часто діапазони використовуються для швидкої генерації списку або іншої колекції чисел. Щоб зробити це, необхідно всього лише передати результат виконання в якості параметра для list. Наступний приклад демонструє створення списку.

data = list (range (10, -25, -5))

print (data)

[10, 5, 0, -5, -10, -15, -20]

***Вивід у зворотньому порядку***

Завдяки реалізації інтерфейсу collections.abc.Sequence ABC, об'єкти класу range можна обробляти різними способами, як списки або кортежі. Таким чином, діапазони мають можливість перевірки входження, пошуку за індексом, зрізу або від'ємної індексації. У наступному прикладі показується вивід списку в Python 3 c range в зворотньому порядку.

data = list (reversed (range (10, -35, -10)))

print (data)

[-30, -20, -10, 0, 10]

Для цього використовується вбудована функція reversed, обробляє послідовність деяких даних і повертає кожен з її елементів в протилежному порядку.

***Відмінності між range і xrange***

Іноді при роботі зі старим кодом, написаними в часи Python 2, можна зустріти метод під назвою ***xrange***. Він дуже схожий на стандартний range. Єдиною відмінністю ***xrange*** від ***range*** є те, що при виклику першого створюється новий об'єкт типу range, а не звичайний список класу ***list***. В такому випадку елементи послідовності не зберігається в пам'яті, а генеруються по ходу виконання програми для економії ресурсів.

В Python 3 функція range була повністю прибрана, а xrange було вирішено перейменувати в range.

Таким чином, тип range (діапазон) має масу шляхів застосування, оскільки дозволяє легко автоматизувати процес заповнення різних наборів чисел. Як правило, для цієї мети застосовується функція range, в якій вказано початковий елемент, межа і крок послідовності. Завдяки реалізації інтерфейсу колекції, діапазони забезпечують перевірку входження, пошук за індексом, зріз і від'ємну індексацію елементів.

1. **Функції map, filter, reduce, zip.**

Як правило, коли говорять про елементах функціонального програмуванні в Python, то подразумевають наступні функції**: lambda, map, filter, reduce, zip**.

В Python **функція map** приймає два аргументи: функцію і аргумент складеного типу даних, наприклад, список. Функція **map** застосовує до кожного елементу списку передану функцію. Наприклад, ви прочитали з файлу список чисел, спочатку всі ці числа мають строковий тип даних, щоб працювати з ними - потрібно перетворити їх в ціле число:

old\_list = ['1', '2', '3', '4', '5', '6', '7']

  new\_list = []

for item in old\_list:

    new\_list.append(int(item))

print (new\_list)

[1, 2, 3, 4, 5, 6, 7]

Той же ефект ми можемо отримати, застосувавши функцію map:

old\_list = ['1', '2', '3', '4', '5', '6', '7']

new\_list = list(map(int, old\_list))

print (new\_list)

[1, 2, 3, 4, 5, 6, 7]

Як бачите такий спосіб займає менше рядків, більш читабельний і виконується швидше. map також працює і з функціями створеними користувачем:

def miles\_to\_kilometers(num\_miles):

    """ Converts miles to the kilometers """

    return num\_miles \* 1.6

mile\_distances = [1.0, 6.5, 17.4, 2.4, 9]

kilometer\_distances = list(map(miles\_to\_kilometers, mile\_distances))

print (kilometer\_distances)

[1.6, 10.4, 27.84, 3.84, 14.4]

А тепер те ж саме, тільки використовуючи lambda вираз:

mile\_distances = [1.0, 6.5, 17.4, 2.4, 9]

kilometer\_distances = list (map (lambda x: x \* 1.6, mile\_distances))

print (kilometer\_distances)

[1.6, 10.4, 27.84, 3.84, 14.4]

Функція map може бути так само застосована для кількох списків, в такому випадку функція-аргумент повинна приймати кількість аргументів, що відповідає кількості списків:

l1 = [1,2,3]

l2 = [4,5,6]

new\_list = list(map(lambda x,y: x + y, l1, l2))

print (new\_list)

[5, 7, 9]

Якщо ж кількість елементів в списках збігатися не буде, то виконання закінчиться на мінімальному списку:

l1 = [1,2,3]

l2 = [4,5]

new\_list = list(map(lambda x,y:  + y, l1, l2))

print (new\_list)

[5,7]

**Функція filter** пропонує елегантний варіант фільтрації елементів послідовності. Приймає в якості аргументів функцію і послідовність, яку необхідно відфільтрувати:

mixed = ['мак', 'просо', 'мак', 'мак', 'просо', 'мак', 'просо', 'просо', 'просо', 'мак']

zolushka = list(filter(lambda x: x == 'мак', mixed))

print (zolushka)

['мак', 'мак', 'мак', 'мак', 'мак']

Зверніть увагу, що функція, передана в filter повинна повертати значення True / False, щоб елементи коректно відфільтрувати.

**Функція reduce** приймає 2 аргументи: функцію і послідовність. reduce () послідовно застосовує функцію-аргумент до елементів списку, повертає середнє арифметичне значення. Зверніть увагу в Python 2.x функція reduce доступна як вбудована, в той час, як в Python 3 вона була переміщена в модуль functools.

Обчислення суми всіх елементів списку за допомогою reduce:

from functools import reduce

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

sum\_all = reduce (lambda x, y: x + y, items)

print (sum\_all)

15

from fu

Обчислення найбільшого елемента в списку за допомогою reduce:

from functools import reduce

items = [1, 24, 17, 14, 9, 32, 2]

all\_max = reduce(lambda a,b: a if (a > b) else b, items)

print (all\_max)

32

Функція zip об'єднує в кортежі елементи з послідовностей переданих в якості аргументів.

a = [1,2,3]

b = "xyz"

c = (None, True)

res = list (zip (a, b, c))

print (res)

[(1, 'x', None), (2, 'y', True)]

Зверніть увагу, що zip припиняє виконання, як тільки досягнутий кінець самого короткого списку.