3. . ЕЛЕМЕНТИ ФУНКЦІОНАЛЬНОГО ПРОГРАМУВАННЯ.

3.1. Що таке функціональне програмування?

Основна перевага використання функцій – це можливість повторного застосування програмного коду, тобто, їх можна викликати багато разів не тільки в тій програмі, де її було визначено, але, можливо, і в інших програмах, іншими користувачами та для інших цілей.

Функціональне програмування (далі ФП) - це стиль програмування,використовує тільки композиції функцій. Іншими словами, це програмування за допомогою виразів, а не імперативних команд.

Функціональне програмування має такі властивості:

- Функції – об'єкти першого класу. Тобто, усе, що можна робити з "даними", можна робити і з функціями.
- Рекурсія як основна структура керування.
- Акцент на обробці списків (Lisp – LISt Processing).
- Функціональні мови унеможливлюють побічні ефекти.
- ФП не схвалює застосування операторів (statements). Замість них – обчислення виразів (тобто функцій з аргументами).
- ФП наголошує на тому, що має бути обчислено, а не як.
- У ФП переважно використовують функції вищого порядку – функції, що оперують функціями

3.2. Функціональна програма.

В математиці функція відображає об'єкти з однієї множини (множини визначення функції) в іншу (множина значень функції). Математичні функції (їх називають чистими) "механічно", однозначно обчислюють результат по заданим аргументам. Чисті функції не повинні зберігати в собі будь-які дані між двома викликами. Чисту функцію можна представити у вигляді чорного ящика, про яку відомо тільки те, що вона робить, але зовсім не важливо, як.

Програми в функціональному стилі конструюються як композиція функцій. При цьому функції розуміються майже так само, як і в математиці: вони відображають одні об'єкти в інші. У програмуванні "чисті" функції - ідеал, не завжди досяжний на практиці. Практично корисні функції зазвичай мають побічний ефект: зберігають стан між викликами або змінюють стан інших об'єктів. Наприклад, без побічних ефектів неможливо уявити собі функції введення-виведення. Власне, такі функції заради цих "ефектів" і використовуються. Крім того, математичні функції легко працюють з об'єктами, які вимагають нескінченного обсягу інформації (наприклад, дійсні числа). У загальному випадку комп'ютерна програма може виконати лише наближені обчислення.

3.3. Функція: визначення і виклик.

Створімо функцію, що виводить числа Фібоначчі до певної межі:

```
def fib(n): # вивести числа Фібоначчі до n
    """Вивести числа Фібоначчі до n."""
    a, b = 0, 1
    while b < n:
        print (b)
        a, b = b, a+b
# Тепер викликаємо щойно задану функцію:
fib(2000)
```

Ключове слово *def* вводить визначення (definition) функції. За ним повинна бути назва функції та оточений дужками список формальних параметрів. Інструкції які утворюють тіло функції починаються з наступного рядка і повинні бути виділені пробілами. Першим, але необов'язковим, рядком функції може бути символьна константа, яка коротко описує функцію. Її називають рядком документації.

Існують спеціальні утиліти, що використовують документаційні рядки для автоматичного створення друкованої чи онлайн документації або для перегляду коду в діалоговому режимі.

Виконання функції вводить новий простір імен, що використовується для локальних змінних функції. Зокрема, всі присвоєння змінним всередині функції зберігають свої значення у локальному просторі імен, тоді як при посиланні на змінну пошук починається у локальному, а потім продовжується у глобальному, і наприкінці – у просторі імен вбудованих ідентифікаторів. Таким чином, глобальні змінні не можуть отримувати нові значення всередині функцій (за винятком якщо вони названі у твердженні global), хоча посилання на них можливе.

Параметри (або аргументи) функції вводяться в простір імен функції при її виклику. Аргументи передаються за значенням, де значення - завжди посилання на об'єкт, а не значення самого об'єкта, тому точніше було б сказати - передача за посиланням. При передачі змінюваного об'єкта будь-які його зміни вседедині викликаної функції стануть видимі в середовищі, що викликало цю функцію, наприклад, додавання нових елементів до списку. Коли одна функція викликає іншу – створюється новий локальний простір імен для цього виклику.

Визначення функції додає назву функції до поточного простору імен. Значення назви функції належить до типу, який ідентифікується інтерпретатором як задана користувачем функція. Це значення може

присвоюватися іншій змінній, що потім теж може використовуватися як функція. Тут показано, як діє загальний механізм перейменування:

```
>>> fib
<function object at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

У Python, як і в C, процедури – це функції, що не повертають жодного значення. Власне, з технічної точки зору, процедури таки повертають певне значення None. Зазвичай, це значення не виводиться інтерпретатором, якщо це єдине можливе значення для виводу.

```
>>> print (fib(0))
None
```

Створення функції, що повертає список чисел Фібоначчі замість виведення їх на друк, досить просте:

```
def fib2(n): # повертає числа Фібоначчі до n
   """Повертає список чисел Фібоначчі до n"""
   result = []
   a, b = 0, 1
   while b < n:
      result.append(b) # див. нижче
      a, b = b, a+b
   return result

f100 = fib2(100) # виклик функції
>>>f100 # вивід результату
 [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Цей приклад також демонструє кілька нових властивостей мови Python:

- Оператор *return* повертає із функції певне значення. Якщо *return* вжито без аргументів, то результатом повернення є *None*. Якщо процедура закінчується сама по собі, то результатом повернення є *None*.
- Інструкція *result.append(b)* викликає метод об'єкта списку *result*. *Метод* - це функція, що "належить" певному об'єкту. Вона викликається у формі *obj.назва_методу*, де *obj* — певний об'єкт (може також бути виразом) і назва_методу — назва методу, визначеного типом об'єкта. Різні типи визначають різні методи. Методи різних типів можуть мати однакову назву, не

спричиняючи при цьому двозначності. Наведений у цьому прикладі метод *append()* визначений для спискових об'єктів; він додає новий елемент в кінець списку. У цьому прикладі це еквівалентно "result = result + [b]", але цей метод є більш ефективним.

Можливо також визначити функцію зі змінною кількістю аргументів. Для цього існує три способи, що можуть сполучуватися.

3.3.1. Стандартні значення аргументів

Найкорисніший спосіб — це визначити типове значення для одного чи кількох аргументів. Це створює можливість виклику функції з меншою кількістю аргументів, аніж задано у визначенні функції. Наприклад:

```
def ask_ok(prompt, retries=4, complaint='Так чи ні, будь-ласка!'):
   while True:
      ok = input(prompt)
      if ok in ('т', 'та', 'так'): return True
      if ok in ('н', 'ні', 'нєт'): return False
      retries = retries - 1
      if retries < 0: raise IOError('затятий користувач')
      print (complaint)
```

Ця функція може викликатися як *ask_ok('Справді закрити програму?')* чи як *ask_ok('Переписати файли?', 2)*.

Цей приклад також ілюструє ключове слово *in*, що дозволяє перевірити чи дана послідовність містить певний елемент.

Стандартні значення обчислюються в момент задання функції відповідно до визначаючого контенту, тому:

```
i = 5
def f(arg=i):
   print( arg)
i = 6
f()
```

Виведе 5.

Важливе зауваження: стандартнне значення обчислюється лише раз. Хоча існує виняток, коли стандартнне значення — змінюваний об'єкт, скажімо, список, словник, реалізація більшості класів. Наприклад, наступна функція акумулює аргументи, що передаються при подальших викликах:

```
def f(a, L=[]):
   L.append(a)
```

```
    return L

print (f(1))
print (f(2))
print (f(3))
```

Виведе:

```
[1]
[1, 2]
[1, 2, 3]
```

Якщо потрібно, щоб стандартне значення було спільним для всіх наступних викликів, можна створити функцію на зразок:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L

print (f(1))
print (f(2))
print (f(3))
```

3.3.2. Ключові аргументи

Функції можуть також викликатися за допомогою ключових аргументів у вигляді "*ключ = значення*". Наприклад:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print ("-- This parrot wouldn't", action)
    print ("if you put", voltage, "Volts through it.")
    print ("-- Lovely plumage, the", type)
    print ("-- It's", state, "!")
```

може викликатися у будь-який вказаний нижче спосіб:

```
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

але такі виклики неправильні:

parrot() # пропущено обов'язковий аргумент

parrot(voltage=5.0, 'dead') # неключовий аргумент передається після ключового

parrot(110, voltage=220) # два значення для одного аргумента

parrot(actor='John Cleese') # невідомий ключ

Взагалі в списку аргументів позиційні аргументи повинні бути розташовані перед ключовими, при цьому ключі повинні бути вибрані з формальних назв параметрів. Чи задані для цього параметра стандартні значення — не важливо. Жоден аргумент не може отримувати значення більш, ніж один раз — формальні назви параметрів, що відповідають позиційним аргументам не можуть використовуватися як ключові слова під час того самого виклику. Ось приклад того, коли помилка відбувається саме через це обмеження:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
 File "&lt;stdin&gt;", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Якщо останній формальний параметр задано у формі **назва, то він отримує словник, що складається з аргументів, чиї ключі відповідають формальним параметрам. Він може сполучатися з формальним параметром у формі *назва. який отримує кортеж, що складається з позиційних аргументів, не включених у список формальних параметрів (аргумент *назва повинен передувати аргументу **назва). Наприклад, функцію, задано таким чином:

```
def cheeseshop(kind, *arguments, **keywords):
   print ("-- Do you have any", kind, '?')
   print ("-- I'm sorry, we're all out of", kind)
   for arg in arguments: print (arg)
   print ('-'*40)
   keys = keywords.keys()
   for kw in keys: print (kw, ':', keywords[kw])

cheeseshop('Limburger', "It's very runny, sir.",
          "It's really very, VERY runny, sir.",
          client='John Cleese',
```

shopkeeper='Michael Palin',
sketch='Cheese Shop Sketch')

Виведе:

-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
----
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch

3.3.3. Списки аргументів довільної довжини

Аргументи передаються за допомогою кортежа. Нуль чи більше звичайних аргументів можуть передувати змінній кількості аргументів.

```
def fprintf(file, format, *args):
   file.write(format % args)
```

Розпакування списків аргументів

Зворотня ситуація трапляється, коли аргументи задані списком чи кортежем, але їх потрібно розпакувати для виклику функції, що потребує окремих позиційних аргументів. Наприклад, вбудована функція *range()* потребує двох окремих аргументів, що вказують на межі послідовності. Якщо вони не задані окремо, виклик функції слід писати з оператором *, що дозволяє розпакувати аргументи, задані списком чи кортежем:

```
>>> range(3, 6) # звичайний виклик з окремими аргументами
[3, 4, 5]
>>> args = [3, 6]
>>> range(*args) # виклик із аргументами, розпакованими зі списку
[3, 4, 5]
```

Лямбда-функції

За популярною вимогою до Python було додано кілька нових властивостей, типових для функціональних мов програмування та мови Lisp. Ключове слово *lambda* дозволяє створювати невеличкі анонімні функції. Ось, наприклад, функція, що повертає суму двох своїх аргументів: "*lambda a, b: a+b*". Лямбда-функції можуть стати в нагоді, коли потрібні об'єкти функцій. Подібно до вкладених функцій лямбда-функції можуть посилатися на змінні із зовнішнього контексту:

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

3.4. Рекурсія.

Кожна з функцій може викликати інші функції, у тому числі звертатися до самої себе. Функцію, яка звертається до самої себе, називають рекурсивною. *Рекурсія* – виклик функції з неї ж самої, безпосередньо (пряма) або через інші функції (складна або непряма): коли функція first викликає функцію second, а та у свою чергу викликає функцію first. Розглянемо рекурсію на прикладі перетворення десяткового числа у список з розрядів його двійкового відображення:

```
def bin(n):
    digits = []
    while n > 0:
        n, d = divmod(n, 2)
        digits = [d] + digits
    return digits

print (bin(69))
```

Тепер перепишемо функцію ***bin*** використовуючи рекурсію:

```
def bin(n):
    if n == 0:
        return []
    n, d = divmod(n, 2)
    return bin(n) + [d]

print (bin(69))
```

Вище показано, що цикл while більше не використовується, а замість нього з'явилося умова закінчення рекурсії: умова, при виконанні якого функція не викликає себе.

Кількість вкладених викликів функції називають глибиною рекурсії.

Обов‘язковим елементом в описі будь-якого рекурсивного процесу є деяке твердження (оператор), який визначає умову завершення рекурсії (іноді його називають опорною умовою). Тут можна задати певне фіксоване значення, яке обов‘язково буде досягнуто під час рекурсивного обчислення, дозволяючи організувати своєчасне призупинення процесу. Крім того, повинен існувати спосіб зображення одного кроку розв‘язання за допомогою іншого, простішого. Кількість рівнів вкладеності не обмежена і може бути досить великою. Кожен виклик рекурсивної функції повинен наближати випадок, який зупиняє рекурсивні виклики, інакше виконання функції ніколи не припиниться через нескінченний ланцюжок рекурсивних викликів. Найпростіший спосіб гарантувати виконання опорної умови є зменшення деякої додатної величини до досягнення деякого «малого» значення.

Особливістю реалізації рекурсивних функцій є те, що у програмі існує тільки один екземпляр коду цієї функції. На кожному рівні рекурсії здійснюється нове звертання до цього коду. Рис. 3.1 ілюструє виконання рекурсивного процесу з трьома рівнями рекурсивного виклику, правда для простоти пояснення вважається, що на кожному рівні створюється новий екземпляр коду. Розглянемо послідовність дій, які виконують у цьому випадку.

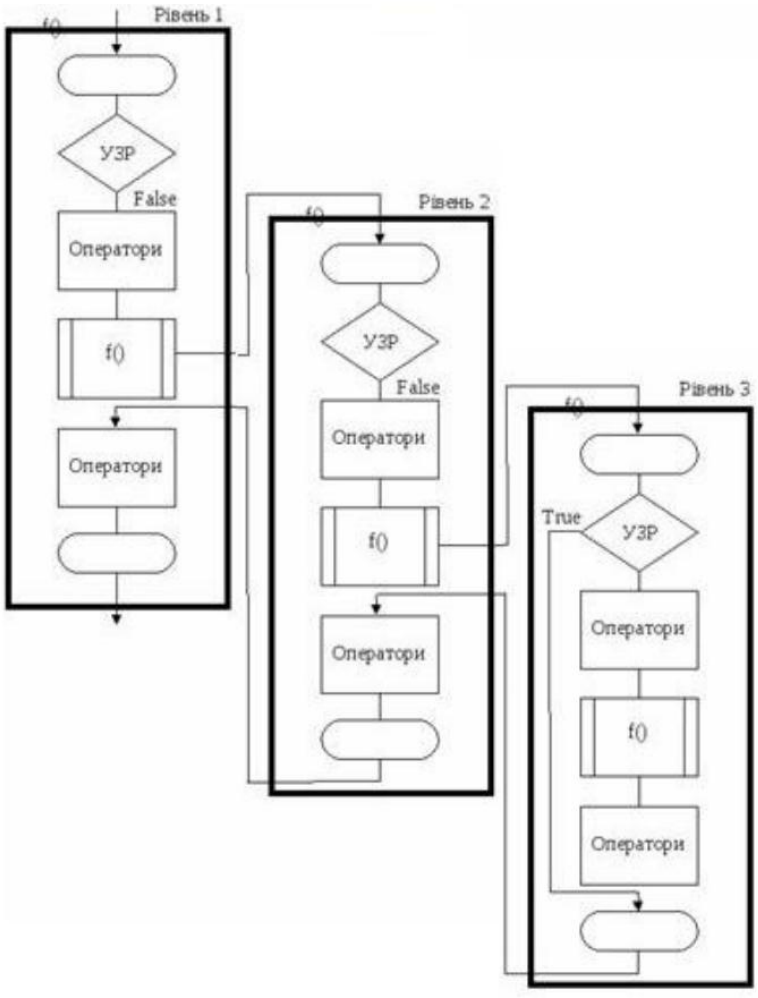

Рис. 3.1 – Схема виконання рекурсивного процесу (УЗР– умова завершення рекурсії)

Виклик функції забезпечує початок виконання її коду. Припускаючи, що на першому рівні рекурсії опорна умова не виконується, маємо виконання операторів, розташованих за опорною умовою, і новий виклик функції.

На цьому виконання функції на першому рівні тимчасово переривається, і розпочинається другий рівень рекурсії. Другий рівень рекурсії (рис.3.1) є аналогом першому. Він також тимчасово переривається з переходом до третього рівня рекурсії. Якщо припустити, що на третьому рівні рекурсії опорна умова виконана, маємо завершення третього рівня з поверненням на другий рівень у точку, де відбулося тимчасове переривання виконання процесу. Далі завершуються обчислення на другому рівні рекурсії з наступним поверненням на перший рівень. Після завершення обчислень на першому рівні, отримуємо остаточний результат.

Рекурсивні функції можуть вимагати великих обсягів пам‘яті для свого виконання. Це пояснюється тим, що, як і у випадку звичайних функцій, локальні змінні створюються в програмному стеку на кожному рівні рекурсивного виклику, що може привести до ситуації, коли обсяг стека вичерпається, оскільки знищення локальних змінних здійснюється тільки при виході з функції.

Будь–яку рекурсивну функцію можна замінити циклом.

Функції, які виконують один рекурсивний виклик на кожній рекурсивній гілці. Структурно рекурсивна функція на верхньому рівні завжди має команду розгалуження у вигляді вибору однієї з двох або більше альтернатив в залежності від умови (умов), яку в цьому випадку доречно назвати «умовою припинення рекурсії». Умова має дві або більше альтернативні гілки, з яких хоча б одна є рекурсивною і хоча б одна – термінальною.

Рекурсивну гілку виконують, коли умова припинення рекурсії має значення «хибність», і містить хоча б один рекурсивний виклик – прямий або опосередкований виклик функцією самої себе.

Термінальну гілку виконують, коли умова припинення рекурсії має значення «істина» (вона повертає деяке значення, не виконуючи рекурсивного виклику). Правильно написана рекурсивна функція повинна гарантувати, що через скінчену кількість рекурсивних викликів буде досягнуто виконання умови припинення рекурсії, в результаті чого ланцюжок послідовних рекурсивних викликів буде перерваний.

Бувають випадки «паралельної рекурсії», коли на одній рекурсивній гілці виконують два або більше рекурсивних виклики. Паралельна рекурсія типова при обробці складних структур даних, таких як дерева. Найпростіший приклад паралельно-рекурсивної функції – обчислення ряду Фібоначчі, де для отримання значення n-го члена необхідно обчислити (n–1)-й і (n–2)-й.

```
def Fib(n):
    if n < 2:
        return n
    else:
        return Fib(n-1) + Fib(n-2)

print(Fib(33))
```

Питання про бажаність використання рекурсивних функцій в програмуванні є неоднозначним. З одного боку, рекурсивна форма може бути структурно простішою і наочнішою, особливо, коли сам алгоритм, по суті є рекурсивним. З іншого боку, рекомендують уникати рекурсивних програм, які призводять (або в деяких умовах можуть призводити) до рекурсії великої глибини. Приклад рекурсивного обчислення факторіала є, скоріше, прикладом того, як не треба застосовувати рекурсію, тому що призводить до досить великої глибині рекурсії і має очевидну реалізацію у вигляді звичайного циклічного алгоритму.