МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ МІСЬКОГО ГОСПОДАРСТВА імені О. М. БЕКЕТОВА

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

до проведення практичних,
самостійної та розрахунково-графічної робіт
з навчальної дисципліни

«СПЕЦІАЛІЗОВАНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ»

(для студентів другого (магістерського) рівня вищої освіти зі спеціальності 193 – Геодезія та землеустрій)

Харків
ХНУМГ ім. О. М. Бекетова
2021

Методичні рекомендації до проведення практичних, самостійної та розрахунково-графічної робіт з навчальної дисципліни «Спеціалізоване програмне забезпечення» (для студентів другого (магістерського) рівня вищої освіти зі спеціальності 193 – Геодезія та землеустрій) / Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова ; уклад. : С. Г. Нестеренко, М. Л. Мироненко, В. В. Касьянов, А. А. Євдокімов. – Харків : ХНУМГ ім. О. М. Бекетова, 2021. – 73 с.

Укладачі: канд. техн. наук, доц. С. Г. Нестеренко,
асист. М. Л. Мироненко,
канд. техн. наук, ст. викл. В. В. Касьянов,
канд. техн. наук, доц. А. А. Євдокімов,

Рецензент
К. А. Мамонов, доктор економічних наук, професор Харківського національного університету міського господарства імені О. М. Бекетова.

Рекомендовано кафедрою земельного адміністрування та геоінформаційних систем, протокол № 1 від 31.08.2021.

ЗМІСТ

ВСТУП

Метою навчальної дисципліни «Спеціалізоване програмне забезпечення» є навчити студентів самостійно будувати програми різної складності мовою програмування Python за допомогою використання структурно-модульного методу програмування.

Основними завданнями вивчення дисципліни «Спеціалізоване програмне забезпечення» є вивчення принципів використання мови програмування Python, а також засвоєння практичних аспектів побудови базових алгоритмів та програм різного рівня складності.

Предметом вивчення дисципліни є теоретичні і практичні основи сучасних технологій програмування мовою Python, методів побудови алгоритмів, використовуваних під час розв’язання прикладних задач в галузі геоінформаційних систем та земельного адміністрування.

Під час вивчення навчальної дисципліни «Спеціалізоване програмне забезпечення» студенти повинні застосовувати мову програмування Python для практичної реалізації задач; програмувати задачі для реалізації ГІС-проєктів; застосовувати оптимальні варіанти під час вирішення визначених завдань та розробляти алгоритми й методики для їх реалізації [1].

Методичні рекомендації призначені для студентів 1 курсу денної та заочної форм навчання другого (магістерського) рівня вищої освіти зі спеціальності 193 – Геодезія та землеустрій.

1 СТРУКТУРА ПРАКТИЧНИХ ЗАНЯТЬ З ДИСЦИПЛІНИ «СПЕЦІАЛІЗОВАНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ»

Змістовий модуль 1 **«Основи програмування на мові Python»**

Тема 1 Вступ до програмування мовою ***Python***.

Мета заняття: ознайомитися з інтерпретатором і середовищем програмування у ***Python***.

Зміст заняття:

1. Установлення інтерпретатора й середовища програмування у ***Python***.

2. Ознайомлення з набором лексичних, синтаксичних та семантичних правил.

3. Написання скриптів у ***Python***.

Тема 2 Типи даних у мові програмування ***Python***.

Мета заняття: ознайомитись з типами даних у мові програмування ***Python***.

Зміст заняття:

1. Цілочисельні і строкові типи даних.

2. Ознайомлення з логічними типами даних і логічними операціями.

3. Ознайомлення з умовними операторами та вкладеними умовними операторами.

4. Застосування конструкції ***if-else***.

5. Цикл while та оператор ***continue*** під час розв'язання задач.

6. Вивчення конструкції дійсних чисел та проблем, які можуть виникати з цим типом даних.

7. Округлення дійсних чисел.

8. Розв'язання задач із застосуванням вивчених конструкцій.

Змістовий модуль 2 **«Особливості реалізації алгоритмів на мові програмування Python»**

Тема 3 Алгоритмічні структури у мові програмування ***Python***.

Мета заняття: ознайомитися з алгоритмічними структурами у мові програмування ***Python*** для побудови структури коду комп'ютерної програми.

Зміст заняття:

1. Структури коду в мові програмування ***Python***.
2. Створення та перевірка умов у мові програмування ***Python***.
3. Розгалуження у мові програмування ***Python***.
4. Створення та перевірка повторень у мові програмування ***Python***.
5. Включення (скорочення синтаксису) у мові програмування ***Python***.
6. Генератори у мові програмування ***Python***.
7. Функції у мові програмування ***Python***.
8. Виняткові ситуації під час обробки помилок.

Тема 4 Модулі та пакети у мові програмування ***Python***.

Мета заняття: ознайомитися з особливостями застосування лазерних сканувальних систем.

Зміст заняття:

1. Імпорт модулів: інструкція import у мові програмування ***Python***.
2. Пакети у мові програмування ***Python***.
3. Стандартна бібліотека мови програмування ***Python***.

Змістовий модуль 3 **«Розробка додатків з використанням мови програмування Python»**

Тема 5 Об'єктно-орієнтований підхід у мові програмування ***Python***.

Мета заняття: вивчити об'єктно-орієнтоване програмування (далі – ООП), як одну з найефективніших методологій створення програмних продуктів.

Зміст заняття:

1. Особливості об'єктно-орієнтованого програмування та об'єкти мови програмування Python.
2. Створення та використання класу у мові програмування ***Python***.
3. Створення класів та екземплярів на мові програмування ***Python***.

4. Наслідування у мові програмування ***Python***.

5. Перевизначення методу у мові програмування ***Python***.

6. Екземпляри, як атрибути у мові програмування ***Python***.

Тема 6 Програмування додатків баз даних на мові програмування ***Python***.

Мета заняття: засвоїти особливості програмування додатків баз даних на мові програмування ***Python***.

Зміст заняття:

1. Групи файлів та особливості роботи з ними.

2. Бази даних у мові програмування ***Python***.

3. Особливості роботи із датою та часом у мові програмування ***Python***.

2 ОРГАНІЗАЦІЯ ТА ЗМІСТ ПРАКТИЧНИХ ЗАНЯТЬ

Для виконання практичних робіт необхідно встановити інтерпретатор мови Python 3 з офіційного сайту: https://www.python.org/downloads/ та середовище програмування JetBrains PyCharm (Community): https://www.jetbrains.com/pycharm/download.

Варто зазначити, що скрипти можна готувати в будь-якому текстовому редакторі, головне, щоб він підтримував підсвічування синтаксису мови програмування ***Python***. Наприклад, підготувати скрипти можна безпосередньо в середовищі розробки ***IDLE*** мови програмування ***Python***.

Програми на мові ***Python*** є звичайні текстові файли, у яких записана послідовність команд. Код легко читається і інтуїтивно зрозумілий.

Наприклад, програма ***Hello, world!*** програма записується в один рядок:

```
print ('Hello, world!')
```

У цій програмі викликається функція друку ***print***, якою, як параметр, передається рядок, що містить фразу ***Hello***, ***world!*** Якщо потрібно задати певний рядок, то необхідно оточити фразу одинарними (') або подвійними (") лапками, інакше вона буде інтерпретуватися, як код на мові ***Python***.

На відміну від рядків, розглянемо цілочисельний тип даних. Наприклад, можна розрахувати результат обчислення арифметичного виразу 2 + 3 і вивести його за допомогою однорядкової програми на мові ***Python***:

```
print (2 + 3)
```

Така програма виведе результат обчислення виразу, який буде дорівнювати 5. Якщо числа 2 і 3 записані в лапках, то вони інтерпретуватися, як рядки, а операція + конкатентувала (склеювала) рядки. Наприклад, код

```
print ( '2' + '3')
```

виведе 23 – рядок, що складається зі «склеєних» символів '2' і '3'.

Функція ***print*** може мати й кілька параметрів, тоді вони будуть виводитися через пробіл, до того ж параметри можуть мати різні типи. Якщо потрібно отримати виведення виду 2 + 3 = 5, то можна використати таку програму:

```
print ('2 + 3 =', 2 + 3)
```

Зверніть увагу, що в рядку '2 + 3 =' немає пробілу після знака =. Пропуск з'являється автоматично між параметрами функції ***print***. Що ж робити, якщо потрібно вивести рядок виду 2 + 3 = 5 (без пробілів)? Для цього знадобиться іменований параметр ***sep*** (separator, роздільник) для функції ***print***. Та рядок, який передається як параметр ***sep***, буде підставлятися замість пробілу, як роздільник. У цьому завданні будемо використовувати порожній рядок, як роздільник. Порожній рядок задається двома лапками:

```
print ('2 + 3 =', 2 + 3, sep = '')
```

Як параметр ***sep***, можна використовувати будь-який рядок, зокрема такий, що складається з декількох символів. Якщо потрібно зробити кілька різних роздільників для різних частин рядків, то необхідно використати кілька функцій ***print***. Наприклад, якщо потрібно вивести рядок виду 1 + 2 + 3 + 4 = 10, то можна застосувати такий код:

```
print (1, 2, 3, 4, sep = '+')
print ('=', 1 + 2 + 3 + 4, sep = '')
```

Однак виведення такого коду буде виглядати, як

```
1 + 2 + 3 + 4
= 10
```

Це обумовлено тим, що після кожної функції ***print*** за замовчуванням здійснюється виведення з нового рядка. Щоб змінити друковане після виведення всього, що містить функція ***print***, можна використати іменний параметр ***end***. Наприклад, у нашому випадку після першого ***print*** ми не хотіли б друкувати нічого. Правильний код виглядатиме так:

```
print (1, 2, 3, 4, sep = '+', end = '')
print ('=', 1 + 2 + 3 + 4, sep = '')
```

Як ***end***, можна використовувати будь-який рядок.

Змінні

У деяких задачах обчислення зручно проводити, використовуючи допоміжні змінні. Наприклад, у шкільних формулах із фізики було зручно обчислювати не

об’ємні вирази, а запам’ятовувати результати обчислення в допоміжних змінних. Для прикладу, обчислимо пройдену відстань за відомим часом і швидкістю:

```
speed = 108
time = 12
dist = speed * time
print (dist)
```

Для надання значення змінної використаємо знак =. Ім’я змінної потрібно записати зліва від знака присвоювання, а арифметичний вираз (у якому можуть використовуватися числа та інші вже задані змінні) – справа. Ім’я змінної має починатися з маленької латинської літери, має бути зрозумілим (англійські слова або загальновживані скорочення) і по довжині не повинно перевищувати 10–15 символів. Якщо логічне ім’я змінної складається з декількох слів, то потрібно записувати його за допомогою ***camelTyping*** (кожне нове слово, крім першого, має записуватися з великої літери).

Арифметичні вирази

Арифметичні вирази використовуються в багатьох операціях додавання «+» і множення «×». Також існує низка інших арифметичних операцій, наведених у таблиці 1.

Таблиця 1 – Арифметичні операції в ***Python***

Оператор	Операція	Вираз	Результат
+	Додавання	7 + 2	9
–	Віднімання	7 – 2	5
/	Ділення	7 / 2	3,5
×	Множення	7 × 2	14
%	Залишок від ділення	7 % 2	1
//	Цілочисельне ділення	7 // 2	3
××	Піднесення до степеня	7 ×× 2	49

Усі операції інфіксні (записуються між операндами), наприклад, для зведення 2 в степінь 3 потрібно написати 2××3.

Рядки також можна зберігати в змінні і використовувати в деякій обмеженій кількості виразів. Зокрема, можна склеювати два рядки за допомогою операції +:

```
goodByePhrase = 'Hasta la vista'
person = 'baby'
print (goodByePhrase + ',' + person + '!')
```

Складати число з рядком (і навпаки) не можна. Але можна використати функцію ***str***, яка за кількістю генерує рядок. ***Str*** – це скорочення від слова ***string***, яке можна перекласти, як «рядок, що є послідовністю символів». Наприклад, завдання для виведення 2 + 3 = 5 можна вирішити і таким способом:

```
answer = '2 + 3 =' + str (2 + 3)
print (answer)
```

Розглянемо задачу, що розв'язується за допомогою арифметичних операцій та демонструє деякі ідеї.

Нехай є два товари, перший з них коштує A гривень B копійок, а другий – C гривень D копійок. Скільки гривень і копійок коштують ці товари разом?

У задачах, де є кілька розмірностей величин (наприклад, гривні і копійки, кілометри і метри, години і хвилини) потрібно переводити все в найменшу одиницю виміру, здійснювати необхідні дії, а потім переводити назад, у потрібні одиниці.

У цій задачі найменшою одиницею є копійки, тому всі ціни потрібно подати в них, потім скласти, а далі перевести результат в гривні і копійки. Код розв'язання буде таким:

```
a = int (input ())
b = int (input ())
c = int (input ())
d = int (input ())
cost1 = a * 100 + b
cost2 = c * 100 + d
totalCost = cost1 + cost2
print (totalCost // 100, totalCost% 100)
```

Можна помножити рядок на ціле невід'ємне число, унаслідок отримаємо вихідний рядок, повторений задану кількість разів:

```
word = 'Bye'
```

```
phrase = word * 3 + '!'
print (phrase)
```

Програми, у яких тільки виводиться значення, але не зчитуються, майже не становлять інтересу для користувачів. Інформація ззовні подається у програми за допомогою функції ***input ()***. Ця функція зчитує рядок з консолі; щоб закінчити введення рядка, потрібно натиснути ***Enter***.

У багатьох задачах потрібно працювати з уведеними числами, а читаються тільки рядки. Щоб перетворити рядок, що складається з цифр (і, можливо, знака "-" перед ними) на ціле число, можна використати функцію ***int*** (скорочення від англійського ***integer***, «ціле число»). Наприклад, розв'язок задачі про складання двох чисел буде виглядати так:

```
a = int (input ())
b = int (input ())
print (a + b)
```

Функцію ***int*** можна застосовувати не тільки щодо результату, що повертається функцією ***input***, а й до довільного рядка.

Логічні вирази

За аналогією з арифметичними виразами використовують логічні вирази, які можуть бути істинними або помилковими. Простий логічний вираз виглядає так: ***<арифметичний вираз> <знак порівняння> <арифметичний вираз>***. Наприклад, якщо маємо змінні ***x*** і ***y*** з якимись значеннями, то логічний вираз ***x+y < 3×y***, як перший арифметичний вираз, має x + y, як знак порівняння – <(менше), а другий арифметичний вираз у ньому – ***3 × y***.

У логічних виразах використовують такі знаки порівнянь (табл. 2).

Таблиця 2 – Знаки порівняння у логічних виразах

Знак порівняння	Опис
<	менше
>	більше
<=	менше або дорівнює
>=	більше або дорівнює
==	дорівнює (рівність)
!=	не дорівнює

У ***Python*** допустимі і логічні вирази, що містять кілька знаків порівняння, наприклад ***x <y <z***. До того ж усі порівняння однаково пріоритетні, що менше, ніж у будь-якій арифметичній операції.

Результат обчислення логічного виразу можна зберігати в змінну, яка буде мати вид ***bool***. Змінні такого виду, як числа і рядки, є незмінними об’єктами.

Рядки також можна порівнювати. До того ж порівняння відбувається в лексикографічному порядку (як впорядковані слова в словнику).

Логічні операції

Щоб записати складний логічний вираз, зазвичай необхідно використати логічні зв’язки "і", "або" і "не". У ***Python*** вони позначаються як ***and***, ***or*** і ***not*** відповідно. Операції ***and*** і ***or*** бінарні, тобто повинні бути записаними між операндами, наприклад x <3 ***or*** y> 2. Операція ***not*** – унарна і повинна записуватися перед єдиним власним операндом.

Усі логічні операції характеризуються нижчою пріоритетністю порівняно з операціями порівняння (а отже і арифметичними операціями). Серед логічних операцій найвищим пріоритетом характеризується операція ***not***, далі ***and*** і найменшим пріоритетом ***or***. На порядок виконання операцій можна впливати за допомогою дужок, як і в арифметичних виразах.

Одним із прикладів використання логічного виразу є перевірка на подільність. Наприклад, щоб перевірити, чи є число парним, необхідно порівняти залишок від ділення цього числа на два з нулем:

```
isEven = number % 2 == 0
n = int(input())
isEven = n % 2 == 0
print(isEven)
```

Результати запуску програми:

```
9
False
або ж
10
True
```

Умовні оператори

Зазвичай логічні вирази застосовуються в умовних операторах. Умовний оператор дозволяє виконувати дії залежно від того, виконана умова чи ні. Записується умовний оператор, як ***"if <логічний вираз>:"***, далі блок команд, який буде виконаний тільки за умови, що логічний вираз набув значення ***True***. Блок команд, який буде виконуватися, виділяється відступом в 4 пробіли (в ***IDE*** можна натискати клавішу ***tab***).

Розглянемо, наприклад, задачу щодо знаходження модуля числа. Якщо число від'ємне, то необхідно замінити його на це число з мінусом. Розв'язок виглядає так:

```
x = int(input())
if x < 0:
    x = -x
print(x)
```

У цій програмі з відступом записаний тільки один рядок: ***x = -x***. Якщо необхідно виконати кілька команд, усіх їх потрібно записати з тим самим відступом. Команда ***print*** записана без відступу, тому вона буде виконуватися в будь-якому разі, незалежно від того, чи була умова в ***if*** істинним значенням чи ні.

Окрім ***if*** можна використовувати оператор ***else*** (інакше). Блок команд, розміщений після оператора, буде виконуватися, якщо умова була помилковою. Наприклад, ту саму задачу про виведення модуля числа можна розв'язати, не змінюючи значення змінної ***x***:

```
x = int(input())
if x >= 0:
```

```
    print(x)
else:
    print(-x)
```

Усі команди, які виконуються в блоці ***else***, потрібно також записати з відступом. ***Else*** повинен розміщуватися відразу за блоком команд ***if***, без проміжних команд, які виконуються безумовно. ***Else*** без відповідного ***if*** не має сенсу.

*Я*кщо після ***if*** записано не логічний вираз, то він буде логічним, якщо від нього була викликано функцію ***bool***. Однак зловживати цим не слід, тому що це погіршує прочитаність коду.

Вкладений умовний оператор

Усередині блоку команд можуть міститися інші умовні оператори. Розглянемо приклад. За заданою кількістю очей і ніг потрібно навчитися відрізняти кішку, павука, морського гребінця і жучка. У морського гребінця буває більше сотні очей, а в павуків їх вісім. Також у павуків вісім ніг, а у морського гребінця їх немає зовсім. У кішки чотири ноги, а у жучка – шість, але очей у обох по два. Розв’язок:

```
eyes = int(input())
legs = int(input())
if eyes >= 8:
    if legs == 8:
        print("spider")
    else:
        print("scallop")
else:
    if legs == 6:
        print("bug")
    else:
        print("cat")
```

*Я*кщо вкладених умовних операторів декілька, то до якого з них належить ***else***, можна зрозуміти за відступом. Відступ у ***else*** повинен бути такий самий, як у ***if***, до якого він належить.

Конструкція «інакше-якщо»

Розглянемо ще один вид завдань. Наприклад, потрібно за введеним числом вивести його текстову назву англійською мовою хоча б для чисел 1, 2 і 3. Як це можна описати? Можна використати конструкцію із кількох ***if***. Тобто, якщо x = 1, друкуємо слово ***One***. Нехай необхідно, крім перевірки 1, 2 або 3, ще й визначити, що введено інше, відмінне від 1, 2 і 3 значення. Тобто існують варіанти, коли необхідно зробити щось конкретне, і варіант за замовчуванням: що робити, якщо жодне з цих значень не спрацьовує. Звісно, можна написати три окремих ***if*** для кожного з варіантів, потім записати складний логічний вираз, що жоден з них не спрацював (усього чотири ***if***). Можна записати простіше: застосувати ***else***, якщо x не дорівнює 1. Перевіримо, чи не дорівнює він 2? Надрукуємо в цій ситуації ***Two***. До цього ***if*** додамо ***else***, у нього ми потрапимо тільки за умови, що x не дорівнює 1 і 2. Може він дорівнювати 3? Розглянемо цей випадок. І, нарешті, додаємо до нього вже ***else***. Цього можна досягти якщо x не дорівнює ні 1, ні 2, ні 3. Друкуємо щось інше (рис. 1).

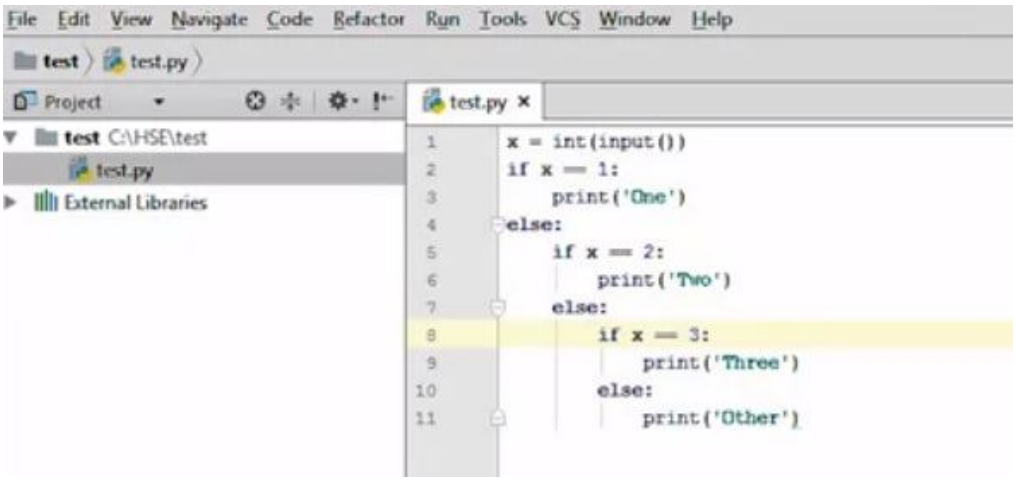

Рисунок 1 – Програмний код

Уявімо, що маємо не три числа, а декілька десятків варіантів. Як бачимо, кожен раз додаються відступи, отже, з часом програма повністю вийде за межі екрану. Читати такий код дуже незручно, тому для цього випадку в ***Phyton*** придумана конструкція ***elif*** – скорочене від ***else*** і ***if***, яка дозволяє не робити додаткового відступу.

У деяких випадках необхідно обирати більше ніж з двох варіантів, які можна обробити за допомогою ***if-else***.

Розглянемо приклад. Необхідно вивести словом назву числа 1 або 2 або повідомити, що це інше число:

```
number = int(input())
if number == 1:
    print('One')
elif number == 2:
    print('Two')
else:
    print('Other')
```

Використовується спеціальна конструкція ***elif***, що означає «інакше, якщо», після якої записується умова. Така конструкція введена в мову ***Phyton***, тому що запис ***if-else*** призведе до збільшення відступу і погіршення прочитаності.

Конструкцій ***elif*** може бути кілька, умови перевіряються послідовно. Як тільки умова виконана, запускається відповідний до цієї умови блок команд і подальша перевірка не виконується. Блок ***else*** є необов'язковим, як і в звичайному ***if***.

Цикл *while*

While перекладається, як «поки», та дозволяє виконувати команди, доти, доки правильна умова. Після закінчення виконання блоку команд, що належать до ***while***, управління повертається на рядок з умовою, і, якщо вона виконана, то виконання блоку команд повторюється, а якщо не виконана, то продовжується виконання команд, записаних після ***while***.

За допомогою ***while*** легко організувати постійний цикл, тому необхідно стежити за тим, щоб у блоці команд відбувалися зміни, які спричинять те, що в якийсь момент умова перестане набувати істинного значення.

Розглянемо приклад. Є число N. Необхідно вивести всі числа по збільшенню від 1 до N. Для вирішення цього завдання потрібно завести лічильник (змінну i), який буде дорівнювати поточному числу. Спочатку це

одиниця. Поки значення лічильника не перевищить N, необхідно виводити його поточне значення і кожен раз збільшувати на одиницю:

```
n = int(input())
i = 1
while i <= n:
    print(i)
    i = i + 1
```

Інструкція для переривання циклу називається ***break***. Після її виконання робота циклу припиняється (ніби умова циклу не була виконана). Осмислене використання конструкції ***break*** можливе, тільки тоді, коли виконана якась умова, тобто ***break*** повинен викликатися тільки всередині ***if*** (міститься всередині циклу). Якщо це можливо, потрібно обходитися без ***break***. Розглянемо приклад постійного циклу, вихід з якого здійснюється за допомогою ***break***. Для цього розв'яжемо завдачу про виведення всіх цілих чисел від 1 до 100. Використовувати ***break*** для цього не потрібно, це тільки приклад:

```
i = 1
while True:
    print(i)
    i = i + 1
    if i > 100:
        break
```

Розрахування суми та оператор *continue*

Команда ***continue*** розпочинає виконання основи циклу заново, починаючи з перевірки умови. Її потрібно використовувати, якщо починаючи з якогось місця в основі циклу і при виконанні будь-яких умов подальші дії небажані.

Наведемо приклад використання ***continue*** (хоча при розв'язанні цієї задачі можна і потрібно обходитися без нього). Дано послідовність чисел, що закінчується нулем. Необхідно вивести всі позитивні числа цієї послідовності. Розв'язання:

```
now = -1
while now != 0:
    now = int(input())
    if now <= 0:
        continue
```

```
print(now)
```

У цьому розв’язку є цікавий момент: перед циклом змінна ініціалізується заздалегідь відповідним значенням. Команда виведення буде виконуватися тільки в тому разі, якщо не виконається умова в ***if***.

Методи *find*, *rfind*, *replace* и *count*. Використання зрізів

Методи – це функції, що застосовуються до об’єктів. Метод викликається за допомогою запису ***ІмяОб’єкта.НазваМетоду*** (параметри). Замість ***ІменіОб’єкта*** можна внести сам об’єкт (рядок). Методи дуже схожі на функції, але дозволяють найкращим чином організувати зберігання й обробку даних. Наприклад, написано структуру даних і потрібно, щоб функція ***len*** повертала довжину структури. Щоб це запрацювало, доведеться звертатися до вихідного коду інтерпретатора ***Phyton*** і вносити зміни у функцію ***len***. Якби ***len*** було методом, то можна б було описати цей метод при створенні структури, не вносячи ніяких змін у код інтерпретатора або стандартної бібліотеки. Отже, методи доцільніше застосовувати для складних структур, наприклад, рядків.

Щодо рядків застосовують різні методи. Розглянемо методи пошуку підрядка в рядку. Метод ***find*** повертає індекс першого входження підрядка в рядок, а якщо його не знайдено – «-1». Наприклад, ***'String'.find (' ing ')*** поверне 3 – індекс, з якого розпочинається входження підрядка ***ing***.

Існують модифікації цих методів з двома параметрами. ***s.find (substring, from)*** буде здійснювати пошук в підрядку ***s [from:]***. Наприклад, ***'String'.find (' ing ', 1)*** поверне 3 (нумерація символів залишається такою, як і в заданому рядку). За аналогією зі зрізами параметри можуть бути й негативними.

Також є модифікації з трьома параметрами: вони шукають підрядок у зрізі s [a : b].

Часто виникає необхідність знайти і вивести всі входження підрядка в рядок, включаючи ті, що накладаються. Наприклад, для рядка ***'ABABA'*** і підрядка ***'ABA'*** відповідь повинна бути така: 0, 2. Розв’язок виглядає так:

```
string = input()
substring = input()
pos = string.find(substring)
while pos != -1:
    print(pos)
    pos = string.find(substring, pos + 1)
```

Або ж

```
s = 'ABABA'
pos = 0
while s.find('ABA', pos) != -1:
    print(s.find('ABA', pos))
    pos = s.find('ABA', pos) + 1
```

Функції

Функції – частини програми, які можна повторно викликати з різними параметрами, щоб не писати багато разів одне й те саме. Функції у програмуванні дещо відрізняються від математичних. У математиці функції можуть тільки набути параметрів і мати відповідь, у програмуванні ж функції можуть, наприклад, нічого не повертати, але щось друкувати. На відміну від математичних функцій, функції на мові ***Python*** не обов'язково повертають значення. Вони можуть тільки виконувати якусь дію. Наприклад, функція ***Print*** стандартна. Вона нічого не повертає, але виконує дію – друкує на екран. Також функції можуть не мати параметрів. Наприклад, відома нам функція ***input*** для читання, ніяких параметрів не набуває, але значення повертає – те, що користувач увів, наприклад, із клавіатури.

Функції надзвичайно корисні, якщо одні й ті самі дії потрібно виконувати декілька разів. Деякі логічні блоки роботи з програмою іноді теж зручно оформляти у вигляді функції. Це пов'язано з тим, що людина може одночасно пам'ятати обмежену кількість речей. Коли програма розростається, відстежити все вже дуже складно. У межах однієї невеликої функції заплутатися набагато складніше – відомо, що вона отримує на вході, що повинна видати, а на інше програма не запрограмована.

У програмуванні також вважається доцільним писати функції, що уміщуються на один екран. У цьому разі можна одночасно переглянути текст всієї функції і не переміщувати текст. Отже, якщо вираз дуже довгий, то

необхідно розділити його на частини так, щоб кожна була логічною (виконувала якусь певну дію, яку можна назвати) і не перевищувала при цьому 10 – 15 рядків.

Готові функції, такі як ***print, len*** і деякі інші, вже використовувалися. Ці функції описані в стандартній бібліотеці або в інших підімкнених бібліотеках. Створимо власні функції.

Розглянемо, як створити свою функцію, на прикладі обчислення факторіала. Текст програми без функції виглядає так:

```
n = int(input())
fact = 1
i = 2
while i <= n:
    fact *= i
    i += 1
print(fact)
```

Обчислення факторіала можна винести у функцію, тоді ця програма буде виглядати так:

```
def factorial(num):
    fact = 1
    i = 2
    while i <= num:
        fact *= i
        i += 1
    return fact

n = int(input())
print(factorial(n))
```

Опис функції має розміщуватися на початку програми. Насправді, він може розташовуватися в будь-якому місці, до першого виклику функції ***factorial***.

Визначення функції має розпочинатися зі слова ***def*** (скорочення від ***define*** – визначити). Далі ім’я функції, після цього в дужках через кому перелічуються параметри (у цій функції всього один параметр). Після закриття – двокрапка.

Команди, що виконуються у функції, необхідно записувати з відступом, як у блоках команд ***if*** або ***while***.

У цій функції ***num*** – це параметр, змість нього підставляється те значення, із яким функція була викликана. Дії всередині функції такі самі, як у звичайній програмі, крім додаткової команди ***return***. Команда ***return*** повертає значення функції (воно повинно бути записано через пропуск після слова ***return***) і припиняє її роботу. Повернене значення підставляється туди, звідки здійснювався виклик функції.

Команда ***return*** може розташовуватися в будь-якому місці функції. Після виконання, функція не використовується. Прослідковується аналогія з командою ***break***, яка застосовується для виходу з циклу.

Налагодження програм

Щоб налагодити програму, якщо сталася помилка, потрібно запустити її в режимі ***debug***. Не просто ***run***, а ***debug*** – налагоджувальний запуск. Вибираємо, що саме потрібно запустити, і отримуємо вираз. Вийшов результат 6. Це означає, що ***debugger*** програма відпрацьована. Припустимо, необхідно прослідкувати, як відбувається цей процес. Розміщуємо ***breakpoint*** на потрібному місці. Для цього натискаємо мишкою поряд з номером рядка. У цьому разі виконання програми на цьому рядку припиниться. Натиснемо ***debug***. Після припинення виконання програми на цьому рядку, замість консолі введення і виведення відкривається вкладка ***debugger***. У вкладці ***debugger*** відображаються всі змінні з їхніми поточними значеннями. Щоб просуватися в програмі, які відрізнятимуться. Існує багато різних ***step*** (***step*** – зробити один крок у виконанні програми) в меню ***Run***: ***step into***, ***step into my code*** і ***step out***. Чим вони відрізняються? ***Step into*** входить до функцій, які викликаються в цьому рядку. ***Step out*** не входить у функцію. Функція виконується відразу. Якщо переконатися, що функція працює правильно, можна використовувати ***step out***, що перейде через цю функцію, підставивши її значення, не заходячи всередину. Якщо ж потрібно прослідкувати, що відбувається всередині функції, необхідно використати ***step into my code***. Тільки ***step into*** буде «провалюватися» в стандартні функції і теж виконувати їх покроково. Стандартні функції

працюють правильно. Проблема може виникнути тоді, коли якщо введені помилкові дані.

Повернення значень

Як було зазначено вище, виконання функції припиняється за командою ***return***. Як приклад, розглянемо функцію пошуку максимуму з двох чисел, які передаються до неї, як параметри:

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

Її можна записати і так:

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

Якщо умова в ***if*** істинна, то виконається команда ***return***, а і виконання функції припинеться.

За допомогою функції ***max2*** можна реалізувати функцію ***max3***, що повертає максимум з трьох чисел:

```
def max3 (a, b, c):
    return max2 (max2 (a, b), c)
```

Ця функція двічі викликає ***max2***: спочатку для вибору максимуму серед чисел ***a*** і ***b***, а потім для вибору максимуму між визначеним значенням і числом ***c***.

Як аргумент функції, може передаватися не тільки змінна чи константне значення, а й результат обчислення будь-якого арифметичного виразу: наприклад, результат, повернутий іншою функцією. Функції ***max2*** і ***max3*** будуть працювати не тільки для чисел, але і для будь-яких об’єктів, що порівнюють, наприклад для рядків.

Повернення логічних значень

Іноді зручно оформляти навіть нескладні обчислення у вигляді функцій, щоб підвищити прочитаність програми. Наприклад, якщо потрібно перевірити число на парність, то доцільно викликати функцію ***isEven (n)***, а не записувати кожен раз ***n% 2 == 0***.

Така функція може виглядати так:

```
def isEven (n):
    return n% 2 == 0
```

Результатом роботи цієї функції буде істина або помилка. Тоді функцію зручно застосувати в ***if***:

```
if isEven (n):
    print ( "EVEN")
else:
    print ("ODD")
```

Якщо наявний складний логічний вираз, то доцільніше оформити його у вигляді функції з виразною назвою – так програму буде легше читати, а ймовірність помилок в ній значно зменшиться.

Усі змінні, використовувані раніше, були глобальними. Глобальні змінні наявні у всіх функціях програми.

Наприклад, код:

```
def f ():
    print a
a = 1
f ()
```

надрукує 1 і виконається без помилок. Мінлива ***a*** – глобальна, тому можна розглядати її значення з будь-якої функції. На момент виклику функції ***f*** змінна ***a*** вже створена, хоча опис функції і випереджає присвоєння.

Якщо ж форматувати змінну всередині функції, то використовувати її поза функцією неможливо. Наприклад, код:

```
def f ():
    a = 1
```

```
f ()
print (a)
```

завершиться з помилкою "***builtins.NameError: name 'a' is not defined***" (змінна ***a*** не визначена). Змінні, значення яких змінюються всередині функції за замовчуванням, вважаються локальними, тобто доступними тільки всередині функції. Після припинення роботи функції змінна знищується.

Таким чином, якщо у функції присвоювалася якась змінна, то ця змінна вважалася локальною. Якщо присвоювання не відбувалося, то змінна вважалася глобальною.

Локальні змінні можна називати тими самими іменами, що й глобальні. Наприклад, виведення коду

```
def f ():
    a = 1
    print (a, end = '')
a = 0
f ()
print (a)
```

виведе "***1 0***". Спочатку відбудеться виклик функції ***f***, у якій буде створено локальну змінну ***a*** зі значенням ***1*** (отримати доступ до глобальної змінної ***a*** з функції неможливо), потім функція припинить роботу і буде виведено глобальну змінну ***a***, зі значення якої не змінилося.

Змінна вважається локальною навіть тоді, коли її присвоєння відбувалося всередині умовного оператора (навіть якщо він ніколи не виконається):

```
def f ():
    print (a)
    if False:
        a = 0
a = 1
f ()
```

Ця програма завершиться з помилкою ***builtins. UnboundLocalError: local variable 'a' referenced before assignment*** – звернення до змінної до ініціалізації. Будь-яке присвоєння значення змінної всередині тіла функції робить змінну локальною.

За допомогою спеціальної команди ***global*** можна змінити значення глобальної змінної функції. Для цього потрібно записати на початку функції слово ***global***, а потім через кому перелічити імена глобальних змінних, які функція може змінити. Наприклад, код

```
def f ():
    global a
    a = 1
    print (a, end = '')
a = 0
f ()
print (a)
```

виведе "***1 + 1***", тому що значення глобальної змінної буде змінено всередині функції. Усі параметри функції – локальні змінні зі значеннями, переданими у функцію. Параметри також можна змінювати, і це ніяк не вплине на значення змінних у тому місці, звідки була викликана функція (якщо тип об'єктів-параметрів був незмінним).

Використовувати глобальні змінні і для читання, і для запису всередині функцій недоцільно. Це пояснюється тим, що в разі використання деяких окремих функцій коду вони не працюють поза програмою, якщо будуть використані глобальні змінні. Тому використовувати глобальні змінні всередині функцій заборонено. Усе потрібне для роботи функції має передаватися, як параметри.

Кортежі

За аналогією з рядками, які можуть зберігати окремі символи, у мові ***Python*** існує тип ***кортеж***, який дозволяє зберігати довільні елементи.

Кортеж може складатися з елементів довільних типів і незмінюваний тип, тобто не можна міняти окремі елементи кортежу, як і символи рядка. Константні кортежі можна створювати в програмі, записуючи елементи через кому і обмежуючи їх дужками. Наприклад, ***testTuple = (1, 2, 3)***. Якщо кортеж є єдиним виразом ліворуч або праворуч від знака присвоювання, то дужки можна

опустити. У всіх інших випадках дужки опускати не варто – це може призвести до помилок.

Багато прийомів і функцій для роботи з рядками підходять і для кортежів, наприклад можна скласти два кортежі:

```
a = (1, 2, 3)
b = (4, 5, 6)
print (a + b)
```

У результаті застосування цієї операції буде виведено ***(1, 2, 3, 4, 5, 6)***. У разі складання створюється новий кортеж, який містить елементи спочатку з першого, а потім із другого кортежу (так само, як і в разі з рядками). Також кортеж можна помножити на число, результат цієї операції аналогічний до множення рядка на число.

Наведемо приклад, коли опускання дужок сричиняє помилку. ***(1, 2) + (3, 4)*** буде давати ***(1, 2, 3, 4)***, а ***1, 2 + 3, 4*** даватиме ***(1, 5, 4)***, оскільки сума буде зрозуміла ***Python,*** як вираз для другого елемента кортежу.

Кортеж можна отримати з рядка, викликавши функцію ***tuple*** з рядка. У результаті кожна буква стане елементом кортежу. До кортежу можна застосовувати функцію ***str***, яка поверне текстове подання кортежу (елементи, перелічені через кому з пропуском і розділені пробілами).

До кортежу можна застосовувати функцію ***len*** і звертатися до елементів за індексом (зокрема за негативним), як і до рядків.

В одному кортежі можуть зберігатися елементи різних типів, наприклад, рядки, числа та інші кортежі упереміш. Наприклад, у кортежі ***myTuple = (('a', 1, 3.14), 'abc', ((1), (2,)))***, ***myTuple [0]*** буде кортежем ***('a', 1, 3.14)***, ***myTuple [1]*** рядком ***'abc'***, а ***myTuple [2]*** кортежем, що складається з числа ***1*** і кортежу з одного елемента ***(2,)***. Числа, записані в дужках, інтерпретуються як числа, за необхідності створити кортеж з одного елемента потрібно після значення елемента поставити кому. Якщо вивести ***myTuple [2] [1]***, то друкується ***(2,)***, а якщо вивести ***myTuple [2] [1] [0]***, то буде надруковано число ***2***.

Розпакуванням називається процес привласнення, у якому кортеж, що складається з окремих змінних, міститься в лівій частині виразу. У такому виразі справа має міститися кортеж тієї самої довжини: наприклад, унаслідок виконання такого коду:

```
manDesc = ( "Ivan", "Ivanenko", 28)
name, surname, age = manDesc
```

У змінній ***name*** задається ***"Ivan"***, у ***surname*** – ***"Ivanenko"***, а в змінній ***age*** – число 28. Англійською розпакування кортежу називається ***tuple unpacking***.

Процес створення кортежу називається упакуванням кортежу. Якщо в одному виразі привласнення відбувається і упакування, і розпакування кортежу, то спочатку виконується упакування, а потім розпакування кортежу. Зокрема внаслідок роботи програми

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

буде виведено ***3 2 1***. Зверніть увагу на те, що функції ***print*** як параметр передається не кортеж, а три цілих числа.

Потрібно розуміти, що запис виду ***(a, b, c) = (c, b, a)*** не є еквівалентним до ланцюжка присвоювання виду ***a = c; b = b; c = a***. Такий ланцюжок присвоювання спричинив те, що в змінних ***a, b, c*** виявилися б значення ***3, 2, 3***.

Функція *range*

У мові ***Python*** є функція ***range***, яка дозволяє генерувати об’єкти типу ***iterable*** (до елементів яких можна отримувати послідовний доступ), що складаються з цілих чисел. ***Iterable*** (ітерований) – об’єкт, що уможливлює почерговий прохід по своїх елементах.

Прикладами типів, що підтримують ітерування по своїх елементах, є послідовності (наприклад: список, рядок, кортеж), а також інші типи (словник, файл).

Для виведення об’єктів типу ***iterable*** використаємо функцію ***tuple***, яка забезпечує утворення кортежу, що складається з усіх елементів ***iterable***, записаних послідовно.

Наприклад, якщо запустити програму

```
print(tuple(range(10))),
```

то буде надруковано (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Функція ***range*** з одним параметром ***n*** генерує ***iterable***, що містить послідовні числа від ***0*** до ***n-1***.

Існує варіант ***range*** з двома параметрами, ***range (from, to)*** згенерує ***iterable*** з усіма числами від ***from*** до ***to-1*** включно:

```
print(tuple(range(9, 25)))
```

Також існує ***range*** з трьома параметрами – ***range (from, to, step)***, який генерує ***iterable*** з числами від ***from***, що не перевищують ***to*** з кроком зміни ***step***:

```
print(tuple(range(9, 25, 3)))
```

Якщо крок від’ємний, то ***from*** повинен бути більшим за ***to***. Наприклад, ***range (10, 0, -2)*** генерує послідовність чисел ***10, 8, 6, 4, 2***. Нуль не входить в цю послідовність.

Параметри ***range*** значною мірою подібні до значень параметрів у зрізах рядків.

Цикл *for*

Для перегляду всіх елементів будь-чого ітерованого – це може бути ***range***, який генерує правило, рядок, або кортеж – існує список ***for***.

Цикл ***for*** дозволяє почергово переглядати елементи будь-чого ітерованого (***iterable*** або ***tuple***). Наприклад, можна переглянути назви кольорів яблук таким способом:

```
for color in ('red', 'green', 'yellow'):
    print(color, 'apple')
```

Унаслідок виконання цієї програми буде надруковано:

```
red apple
green apple
yellow apple
```

На місце змінної ***color*** будуть почергово підставлятися значення з кортежу. Загалом цикл ***for*** виглядає так: ***for ім'яЗмінної in щосьІтероване***.

Усі дії, виконувані у ***for***, необхідно виділяти відступом, як і в ***if*** або ***while***. Роботу циклу ***for*** можна припинити за допомогою команди ***break*** або можна перейти до наступної ітерації за допомогою ***continue***. Ефект від цих команд такий самий, що й під час роботи з циклом ***while***.

Здебільшого ***for*** використовується з функцією ***range***. Наприклад, за допомогою ***for*** можна надрукувати непарні числа від 1 до 100:

```
for i in range(1, 100, 2):
    print(i)
```

Усередині ***for*** може розташуватися й інший ***for***. Так виглядає код для виведення таблиці множення всіх чисел від 1 до 10:

```
for i in range(1, 11):
    for j in range(1, 11):
        print(i * j, end=' ')
    print()
```

У разі використання функції ***range*** у ***for*** не потрібно перетворювати ***iterable*** на ***tuple***. Це пов'язано з тим, що ***for*** прагне отримувати послідовний доступ, який дає ***iterable***. ***Tuple*** може набагато більше, але в цьому випадку його використання призведе до зайвих витрат часу і пам'яті.

Списки

Список у ***Python*** є аналогом масивів в інших мовах програмування.

Список – це набір посилань на об'єкти (як і кортеж), проте він змінний.

Константні списки записують у квадратних дужках, усе інше в них виконується аналогічно до кортежу. Наприклад, можна створити список з числами від 1 до 5: ***myList = [1, 2, 3, 4, 5]***.

Списки і кортежі легко перетворюються один на одного. Для перетворення списку на кортеж потрібно використати вже відому функцію ***tuple***, а для перетворення кортежу на список потрібна функція ***list***. Функцію ***list*** можна застосувати й до рядка. Унаслідок цього отримаємо список, елементами якого буде літера з рядка. Наприклад, ***list ('abc')*** буде виглядати як ***['a', 'b', 'c']***.

До списків також можна застосувати функцію ***len*** і зрізи, які працюють, як у кортежі.

Головною відмінністю списку від кортежу є змінюваність. Тобто певний елемент списку можна змінити (він може розташовуватися в лівій частині операції присвоювання).

Наприклад, унаслідок виконання коду

```
myList = [1, 2, 3]
myList[1] = 4
print(myList)
```

буде надруковано [1, 4, 3].

Змінити символ (або елемент в кортежі) можна, зробивши два зрізи і конкатенацію першої частини рядка, нового символу й «хвоста» рядка. Це дуже повільна операція, час її виконання пропорційний до довжини рядка. Заміна елемента в списку здійснюється за O (1), тобто не залежить від довжини списку.

Методи роботи зі списками

До змінних типу «список» можна застосувати певні методи. Методи, що не змінюють список і повертають значення:

– ***count (x)*** підраховує число входжень значення x у список. Виконується за час O (N):

```
a=[1, 2, 3, 4, 2, 2, 3]
print(a.count(2))
```

Отримаємо 3;

– ***index (x)*** знаходить позицію першого входження значення x у список. Виконується за час O (N);

– ***index (x, from)*** знаходить позицію першого входження значення ***x*** у список починаючи з позиції ***from***. Виконується за час O (N);

Методи, що не повертають значення, але змінюють список:

– ***append (x)*** додає значення x у кінець списку:

```
a=[1, 2, 3, 4, 2, 2, 3]
a.append(5)
print(a)
```

Отримуємо [1, 2, 3, 4, 2, 2, 3, 5];

– ***extend (otherList)*** додає весь вміст списку ***otherList*** у кінець списку. На відміну від операції +, змінює об'єкт до якого застосований, а не створює новий:

```
a=[1, 2, 3, 4, 2, 2, 3]
b=[4, 5]
a.extend(b)
print(a)
```

Отримаємо [1, 2, 3, 4, 2, 2, 3, 4, 5]. Замість використання ***extend*** можна застосувати складання списків, як і інших об'єктів. Дійсно, отримаємо той самий результат:

```
a=[1, 2, 3, 4, 2, 2, 3]
b=[4, 5]
#a.extend(b)
a=a+b
print(a)

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

– ***remove (x)*** видаляє перше входження числа ***x*** у список. Виконується за час O (N):

```
a=[1, 2, 3, 4, 2, 2, 3]
a.remove(2)
print(a)
```

Отримаємо [1, 3, 4, 2, 2, 3]. Як наслідок, зникла тільки перша двійка, тоді як останні – наприкінці списку, залишилися;

– ***insert (index, x)*** вставляє число ***x*** у список так, що воно опиняється на позиції index. Число, що розміщувалось на позиції ***index*** і всі числа правіше від нього, зсуваються на один вправо. Виконується за час O (N):

```
a=[1, 2, 3, 4, 2, 2, 3]
a.insert(2, 10)
print(a)
```

Отримаємо [1, 2, 10, 3, 4, 2, 2, 3]. Зверніть увагу, що ***insert*** працює залежно від довжини списку, тобто виконується копіювання всіх елементів, які там містяться, звільняється місце і вставляється необхідне число ***x***. Тобто операція досить складна, і зловживати ***insert*** всередині довгого списку не варто, оскільки операція буде виконуватися дуже повільно. Можливо, варто застосувати більш ефективний алгоритм, коли додавання необхідне тільки наприкінці, наприклад за допомогою ***append;***

– ***reverse ()*** розгортає список (змінює значення за посиланням, а не створює новий список, як ***myList [:: - 1])***. Виконується за час O (N).

Методи, які повертають значення і змінюють список:

– ***pop ()*** повертає останній елемент списку і видаляє його:

```
a=[1, 2, 3, 4, 2, 2, 3]
a.pop()
print(a)
```

Отримаємо [1, 2, 3, 4, 2, 2]. Метод ***pop*** без параметрів видаляє останній елемент списку. Це можна зробити дуже швидко, за одну операцію;

– ***pop (index)*** повертає елемент списку на позицію ***index*** і видаляє його. Виконується за час O (N).

Можна застосувати ***pop*** з параметром, який укаже, з якої позиції видалити елемент. Наприклад, з третьої позиції, де розміщувалася четвірка:

```
a=[1, 2, 3, 4, 2, 2, 3]
a.pop(3)
print(a)
```

Отримаємо [1, 2, 3, 2, 2, 3].

Операція ***pop*** з параметром – досить повільна операція, окрім того усі елементи копіюються, переставляються на один, і після цього останній видаляється. Тому, ***pop*** необхідно по змозі уникати й застосовувати алгоритм, який видаляє тільки останні елементи.

Сортування списків

У програмуванні зручніше працювати з відсортованими даними. У мові ***Python*** існує можливість впорядкувати списки двома способами. Розглянемо їх

на прикладі розв'язання простої задачі щодо сортування послідовності чисел по неспаданню (як по зростанню, але, можливо, з однаковими числами). Ось перший спосіб упорядкувати його:

```
myList = list (map (int, input (). split ()))
myList.sort ()
print ( ' ' .join (map (str, myList)))
```

У цьому прикладі використовується метод ***sort,*** застосований до списку. Цей метод змінює вміст списку – після застосування методу ***sort*** елементи в списку стають впорядкованими. Такий метод доцільний тільки для об'єктів типу список, його не можна застосувати до кортежу або ***iterable*** чи рядка.

Другий спосіб полягає в застосуванні функції ***sorted***, яка повертає відсортований список, але не змінює значення параметра:

```
myList = list (map (int, input (). split ()))
sortedList = sorted (myList)
print ( ' ' .join (map (str, sortedList)))
```

Використання функції ***sorted*** виправдано в разі, якщо вихідні дані потрібно зберегти в незмінному вигляді з певною метою. Наприклад, ***sorted*** можна використовувати всередині функції для створення відсортованої копії, щоб не змінювати переданий список.

Щоб відсортувати список по незростанню (зменшенням), необхідно передати в метод або функції іменований параметр ***reverse***. Наприклад, це буде виглядати, як ***myList.sort (reverse = True)*** або ***sorted (myList, reverse = True):***

```
myList = list(map(int, input().split()))
b=sorted (myList, reverse = True)
print(*b)
```

Функція ***sorted*** може приймати як параметр не тільки список, але й будь-що інтегроване: кортежі, ***iterable*** або рядки:

```
print (sorted ((1, 3, 2)))
print (sorted (range (10, -1, -2)))
print (sorted ( "cba"))
```

До того ж ***sorted*** завжди повертає список, тобто виведення цієї програми буде таким:

```
[1, 2, 3]
[0, 2, 4, 6, 8, 10]
[ 'a', 'b', 'c']
```

Сортування можна застосовувати до списків, усі елементи яких можна порівняти. Зазвичай це однорідні списки (що складаються з елементів одного типу) або, в окремих випадках, цілі й дійсні числа упереміш.

Задача. Відсортуйте даний масив, використовуючи вбудоване сортування.

Формат введення. Перший рядок вхідних даних містить кількість елементів у масиві N, $N \leq 10^5$. Далі йде N цілих чисел, що не перевищують за абсолютною величиною 10^9.

Формат виведення. Виведіть числа в порядку неспадання.

Приклади для тестування

Тест 1
Вхідні дані:
```
1
1
```
Виведення програми:
```
1
```

Тест 2
Вхідні дані:
```
2
3 1
```
Виведення програми:
```
1 3
```
Тест 3
Вхідні дані:
```
5
5 4 3 2 1
```
Виведення програми:
```
1 2 3 4 5
```

Розв'язок задачі:

```
n = int(input())
myList = list(map(int, input().split()))
myList.sort()
print(' '.join(map(str, myList)))
```

Іменований параметр *key*

Нехай для кожної людини задані її зріст і ім'я, необхідно впорядкувати список людей по зростанню, а в разі однакового зросту – в алфавітному порядку. При вирішенні цього завдання достатньо зберігати опис кожної людини у вигляді кортежу, де першим елементом буде зріст, а другим – прізвище.

Розглянемо приклад. Для кожної людини задані зріст і ім’я. Необхідно впорядкувати їх за зростанням, а в разі однакового зросту – в алфавітному порядку:

```
n = int (input ())
peopleList = []
for i in range (n):
    tempManData = input (). split ()
    manData = (int (tempManData [0]), tempManData [1])
    peopleList.append (manData)
peopleList.sort ()
for manData in peopleList:
    print ( ''.join (map (str, manData )))

n = int(input())
peopleList = []
for i in range(n):
    tempManData = input().split()
    manData = (int(tempManData[0]), tempManData[1])
    peopleList.append(manData)
peopleList.sort()
for manData in peopleList:
    print(''.join(map(str, manData)))
```

Дані для тестування:

```
6
172 Ivan
168 Andrij
191 Vlad
157 Anna
169 Elizaveta
169 Stas
```

Виведення:

```
157Anna
168Andrij
169Elizaveta
169Stas
172Ivan
191Vlad
```

У цьому прикладі складено кортеж, який містить параметри порівнюваних людей в потрібному порядку. Здебільшого виникають неприємні ситуації. Розглянемо ту саму задачу, але людей потрібно впорядкувати за спаданням зросту, а в разі однакового зросту вони, як і вище, – за алфавітом. Просте використання ***reversed = True*** не приведе до бажаного результату: люди з однаковим зростом розміщуватимуться неправильно.

Можна схитрувати і перетворити зростання на негативне число, модуль якого буде дорівнювати вихідному зросту. Після цього список можна впорядкувати за зростанням – найвищі люди матимуть найменший негативний «зріст», за яким відбувається порівняння. Перед виведенням необхідно перетворити зростання на позитивне число:

```
n = int(input())
peopleList = []
for i in range(n):
    tempManData = input().split()
    manData = (-int(tempManData[0]), tempManData[1])
    peopleList.append(manData)
peopleList.sort()
for badManData in peopleList:
    manData = (- badManData[0], badManData[1])
    print(''.join(map(str, manData)))
```

```
n = int(input())
peopleList = []
for i in range(n):
    tempManData = input().split()
    manData = (-int(tempManData[0]), tempManData[1])
    peopleList.append(manData)
peopleList.sort()
for badManData in peopleList:
    manData = (- badManData[0], badManData[1])
    print(''.join(map(str, manData)))
```

Протестуємо:

```
6
172 Ivan
168 Andrij
191 Vlad
157 Anna
169 Elizaveta
169 Stas
```

Виведеня:

```
191Vlad
172Ivan
169Elizaveta
169Stas
168Andrij
157Anna
```

Цей код незрозумілий і недоцільний.

Параметр *key* у функції *sort*

Для реалізації нестандартних угруповань краще НЕ спотворювати вихідні дані, а використовувати параметр ***key***, що передається у функцію сортування.

Значенням цього параметра повинна бути функція, яка застосовується до кожного елемента списку, а порівняння елементів відбувається за значенням цієї функції (воно називається ключем).

Розглянемо такий приклад. Необхідно впорядкувати введені рядки по довжині, а в разі однакової довжини залишити їх в тому самому порядку, що й у вхідному файлі. Наприклад, для вхідних рядків ***"c", "abb", "b"*** правильною відповіддю має бути ***"c", "b", "abb"*** (***"c"*** розміщується за ***"b"***, оскільки їхня довжина однакова, а ***"c"*** у вхідних даних розміщується попереду ***"b"***).

Сортування, використовуване в ***Python***, володіє властивістю стійкості (***stable***), тобто для елементів з рівним ключем зберігається їх взаємний порядок.

Розв'язкам цієї задачі буде таким:

```
n = int(input())
strings = []
for i in range(n):
    strings.append(input())
print('\n'.join(sorted(strings, key=len)))
```

Протестуємо:

```
3
c
abb
b
```

Виведення:

```
c
b
abb
```

Як ще один приклад, розглянемо задачу про сортування точок на площині, заданих парою цілих координат x і y по не зменшенню відстані від початку координат. У цьому випадку як опцію для генерації ключа, за яким будуть порівнюватися елементи, запишемо функцію, яка буде повертати квадрат відстані від точки до початку координат. Квадрат відстані

використовується для того, щоб залишатися в цілих числах і позбутися необхідності вважати квадратний корінь (повільно і неточно):

```
def dist(point):
    return point[0] ** 2 + point[1] ** 2


n = int(input())
points = []
for i in range(n):
    point = tuple(map(int, input().split()))
    points.append(point)
points.sort(key=dist)
for point in points:
    print(' '.join(map(str, point)))
```

Протестуємо:

```
3
1 1
10 1
5 5
```

Виведення:

```
1 1
5 5
10 1
```

Кожен елемент списку – кортежі із двох чисел. Саме такого параметру набуває ця функція. Можна було б використати функцію ***hypot*** з бібліотеки ***math***, щоб не писати власну функцію підрахунку ключа, однак це неможливо – у ній очікується на вхід два числових параметри, а не кортеж.

«Структури» *в Python*

Для зберігання складних записів у багатьох мовах є спеціальні типи даних, такі як ***struct в C ++*** або ***record*** в ***Паскалі***.

Змінна типу «структура» містить декілька іменованих полів. Наприклад, як у задачі сортування людей за зменшенням зросту, зручно зберігати опис кожної людини у вигляді структури з двома полями: зріст і ім’я.

У чистому вигляді типу даних «структура» в стандарті мови ***Python*** немає. Є декілька способів реалізувати аналог структур: ***namedtuple*** з бібліотеки ***collections***, використання словників або класів, як структур.

Розглянемо останній спосіб.

Нагадаємо умову завдання: людей потрібно впорядкувати за спаданням зросту, але якщо зріст однаковий – за прізвищем. Розв'язок з використанням класів, як структур, буде виглядати так:

```
class Man:
    height = 0
    name = ''


def manKey(man):
    return (- man.height, man.name)


n = int(input())
peopleList = []
for i in range(n):
    tempManData = input().split()
    man = Man()
    man.height = int(tempManData[0])
    man.name = tempManData[1]
    peopleList.append(man)
peopleList.sort(key=manKey)
for man in peopleList:
    print(man.height, man.name)
```

Протестуємо:

```
6
172 Ivan
168 Andrij
191 Vlad
157 Anna
169 Elizaveta
169 Stas
```

Виведення:

```
191 Vlad
172 Ivan
169 Elizaveta
169 Stas
168 Andrij
157 Anna
```

Щоб користуватися класами, як структурами, створимо новий тип даних – ***Man***. В описі класу перелічимо імена всіх полів і їхні значення по замовчуванню.

Надалі можна створювати об'єкти класу ***Man*** (це робиться рядком ***man = Man ()***, які спочатку ініціалізують свої поля значеннями по замовчуванню.

Доступ до полів класу здійснюється через крапку.

Функція порівняння приймає об’єкт класу і генерує ключ, за яким ці об’єкти будуть порівнюватися при сортуванні.

Використовувати структури для опису складних об’єктів доцільніше, ніж використовувати кортежі. Якщо параметрів більше двох, використання кортежів ускладнює прочитання для читача і того, хто створює код, оскільки неможливо зрозуміти, що зберігається в ***badNamedTuple [13]*** і не важко зрозуміти що зберігається в ***goodNamedStruct.goodNamedField***.

Лямбда-функції

У окремих випадках функції, які використовуються для отримання ключа сортування, такі прості, що НЕ потрібно оформляти їх стандартно, а одразу написати їх, не надаючи імені.

Це можна здійснити з допомогою лямбда-функцій, які можуть замінити функції, що містять в тілі тільки оператор ***return***. Запис лямбда-функції, що підносить число в квадрат, може виглядати так:

```
lambda x: x ** 2,
```

що еквівалентно звичному запису функції:

```
def sqr (x):
    return x ** 2
```

Відмінність лямбда-функції полягає в тому, що вона не має імені і, отже, викликати її по імені неможливо. Єдиним застосуванням лямбда-функцій може слугувати їхня передача, як параметра в такі функції як ***sort*** або ***map***. Наприклад, за допомогою лямбда-функції можна вивести список квадратів усіх чисел від 1 до 100 в один рядок:

```
print(' '.join(map(lambda x: str(x ** 2), range(1, 101))))
```

У цій програмі лямбда-функція приймає як параметр число, а повертає строкове надання його квадрата.

Повернімося до задачі сортування точок за віддаленістю від початку координат. Цю задачу також можна розв’язати за допомогою лямбда функції:

```
n = int(input())
points = []
```

```
for i in range(n):
    point = tuple(map(int, input().split()))
    points.append(point)
points.sort(key=lambda point: point[0] ** 2 + point[1] ** 2)
for point in points:
    print(' '.join(map(str, point)))
```

Протестуємо:

```
3
1 1
10 1
5 5
```

Виведення:

```
1 1
5 5
10 1
```

Лямбда-функція може набувати кількох параметрів (тоді після слова ***lambda*** потрібно записати їхні імена через кому), однак при використанні їх в ***sort*** або ***map*** зазвичай застосовується один параметр.

У мові ***Python*** функція також є об'єктом і можемо створити посилання на об'єкт типу «функція». Наприклад, два записи функції зведення в квадрат еквівалентні:

```
def traditionalSqr(x):
    return x ** 2


lambdaSqr = lambda x: x ** 2
print(traditionalSqr(3))
print(lambdaSqr(3))
```

Виведення:

```
9
9
```

Такий підхід дозволяє перевикористати лямбда-функції, але переважно варто користовувати стандартне оголошення функції – це спрощує читання і налагодження програми.

Множини і хеш-функції

У мові ***Python*** множини означають те саме, що й у математиці: набір об'єктів без певного порядку. До множини можна додавати і видаляти з неї об'єкти, перевіряти приналежність об'єкта множині і переглядати всі об'єкти

множини.

Також над множинами можна здійснювати групові операції, наприклад, перетинати і об’єднувати дві множини.

Перевірка приналежності елемента множині, а також операції видалення і додавання елементів, здійснюються за O (1) (щоб зберегти елементи в списку і перевірити приналежність елемента списку, потрібно O (N) операцій, де N – довжина списку).

Такої швидкості досягають шляхом використання хеш-таблиць. Хеш-таблиця – це масив досить великого розміру (назвемо цей розмір K). Кожен незмінний об’єкту можна співставити (за деяким правилом) з числом M від 0 до K і розмістити цей об’єкт у клітинці списку з індексом M. Наприклад, для цілих чисел таким правилом співставлення може бути підрахунок залишку від ділення цілого числа на K. Операція взяття залишку і буде хеш-функцією.

Якщо потрібно перевірити, чи належить деяке число множині, треба порахувати хеш-функцію від нього і перевірити чи міститься вона в клітинці з індексом, що дорівнює результату обчислення хеш-функції, потрібний об’єкт чи ні. Для інших типів даних можна застосувати такий підхід: будь-який об’єкт, так чи інакше, є послідовністю байт. Будемо інтерпретувати цю послідовність байтів, як число, і підрахуємо хеш-функцію для цього числа.

Зрозуміло, може виявитися, що кілька об’єктів дають один і той самий хеш (відображення між величезною кількістю різних об’єктів і незначним розміром множини допустимих хешів НЕ може бути біективним). Такі проблеми можна вирішити, не погіршуючи асимптотичну складність.

Оскільки, наприклад, числа можуть бути досить довгими, то операція підрахунку хеш-функції при кожній операції з цим об’єктом у множині може бути дуже повільною. Тому кожен незмінний об’єкт в ***Python*** має заздалегідь нарахований хеш, який підраховується один раз при його створенні. До речі, за допомогою цих самих хешів можна зрозуміти, чи є вже об’єкт в пам’яті, і не створювати нових об’єктів, а тільки надавати ще одне посилання на вже наявний об’єкт.

Змінювані типи, такі як список, не мають заздалегідь розрахованих хешів. Зміна лише одного елемента в списку спричинила б повний перерахок хешу для всього списку, що катастрофічно сповільнило б роботу зі списками. Тому змінювані об'єкти не мають хешу і НЕ можуть додаватися в множину.

Сама множина також є змінюваним об'єктом і не може бути, наприклад, елементом іншої множини.

Існують також незмінювані множини, які створюються за допомогою функції ***frozenset***.

Створення множин

Множину в тілі програми можна створити за допомогою запису елементів через кому у фігурних дужках:

```
mySet = {3, 1, 2}
print (mySet)
```

Виведення за допомогою ***print*** здійснюється в тому самому форматі. Порядок елементів у множині може бути випадковим, оскільки хеш-функція НЕ гарантує, що якщо A > B, то h (A) > h (B).

Якщо під час завдання множини наявні декілька однакових елементів, то вони потраплять у множину в єдиному екземплярі:

```
firstSet = {1, 2, 1, 3}
secondSet = {3, 2, 1}
print (firstSet == secondSet)
```

Робота з множинами

1. *Робота з елементами множин.*

Щоб створити порожню множину, потрібно написати:

```
emptySet = set ()
```

Додавання елемента в множину здійснюється за допомогою методу ***add***. Якщо елемент уже був у множині, то вона НЕ зміниться, наприклад:

```
mySet = {1, 2, 5, 11, 13}
mySet.add(17)
print(*mySet)
```

Виведення:

```
1 2 5 11 13 17
```

Переглянути елементи множини можна за допомогою ***for*** (***for*** можна застосувати до будь-яких інтегрованих об’єктів):

```
mySet = {1, '2', 2, '1'}
for elem in mySet:
    print(elem, end=' ')
```

Виведенням такої програми буде " 1 1 2 2 ", хоча впорядкованість є тільки випадковістю.

Щоб перевірити, чи входить чи елемент X в множину A досить написати ***X in A***. Результатом цієї операції буде ***True*** або ***False***. Щоб перевірити, чи елемент НЕ належить до множини, можна писати not X in A, або, що більш звично ***X not in A***.

```
mySet = {1, 2, 3}
if 1 in mySet:
    print('1 in set ')
else:
    print('1 not in set ')
x = 42
if x not in mySet:
    print('x not in set ')
else:
    print('x in set ')
```

Виведення цієї програми:

```
1 in set
x not in set
```

Щоб видалити елемент з множини, можна використати один із двох методів: ***discard*** або ***remove***. Якщо елемент у множині не виявиться, то ***discard*** не змінить стану множини, а ***remove*** випаде з помилкою:

```
mySet = {1, 2, 5, 11, 13}
mySet. remove (13)
print(*mySet)

mySet = {1, 2, 5, 11, 13}
mySet. discard (17)
print(*mySet)
```

Групові операції із множинами

У ***Python*** можна працювати не лише з окремими елементами множин, але й із множиною загалом. Наприклад, для множин визначені такі операції (табл. 3).

Таблиця 3 – Операції із множинами

Операція	Опис
A | B	Об’єднання множин
A & B	Перетин множин (символ має назву «персант»)
A - B	Множина, елементи якої входять в A, але не входять у B
A ^ B	Елементи входять в A | B, але не входять в A & B

```
a = {1, 2, 3, 4}
b = {1, 3, 5}
print(a|b)
print(a&b)
print(a-b)
print(a^b)
```

Виведення:

```
{1, 2, 3, 4, 5}
{1, 3}
{2, 4}
{2, 4, 5}
```

Унаслідок цих операцій створюється нова множина, проте для неї визначено і скорочений запис : | =, & =, - = і ^ =. Такі операції змінюють множину, що міститься зліва від знака операції. Для множин також визначені операції порівняння (табл. 4).

Таблиця 3 – Операції над множинами

Операція	Опис
A == B	Усі елементи співпадають
A! = B	Є різні елементи
A <= B	Усі елементи A входять у B
A <B	A <= B і A! = B

```
a = {1, 2, 3, 4}
b = {1, 3}
print(a==b)
print(a!=b)
```

Виведення:

```
False
True
```

Порівняння:

```
a = {1, 2, 3, 4}
b = {1, 3}
print(a>b)
print(a<b)
```

Виведення:

```
True
False

a = {1, 2, 3, 4}
b = {1, 2, 3, 4}
print(a>=b)
print(a<=b)
```

Виведення:

```
True
True
```

Також визначено операції > і> =. Усі групові операції і порівняння проводяться із множинами за час, пропорційний до кількості елементів у множині.

Словники

Зазвичай виникає необхідність співставити ключ значення. Наприклад, в англо-українському словнику англійське слово співставиться з одним або декількома російськими словами. Англійське слово – ключ, а російські – значення.

У мові ***Python*** є структура даних словника, яка дозволяє реалізовувати подібні операції. До того ж об'єкти-ключі унікальні, і кожен із них має деякий об'єкт-значення. Обмеження щодо ключів такі самі, що й до елементів множини, тоді як значення можуть бути і змінюваними.

Словник є множиною, де кожен елемент-ключ співставляється з об'єктом-значенням.

Створити словник у вихідному тексті програми можна, записавши у фігурних дужках пари ключ – значення через кому, усередині пари ключ відділяється від значення двокрапкою:

```
countries = { ' Ukraine ': ' Europe ', 'Germany ': ' Europe ', ' Australia ': ' Australia '}
```

Додавати пари ключ – значення в словник дуже просто: це робиться за аналогією зі списками:

```
sqrs = {}
sqrs [1] = 1
```

```
sqrs [2] = 4
sqrs [10] = 100
print ( sqrs )
```

Порожній словник можна створити, написавши порожні фігурні дужки (це буде словник, а не множина).

Словник також можна конструювати з інших об'єктів за допомогою функції ***dict***:

```
myDict = dict ([[ 'key1', 'value1'], ( 'key2', 'value2')])
print ( myDict )
```

На вхід функції повинен подаватися ***iterable***, кожен елемент якого, зі свого боку, є ***iterable*** з двома елементами – ключем і значенням.

Дізнаватися значення за ключем можна також за допомогою запису ключа після імені словника в квадратних дужках:

```
phones = { ' police ': 102, ' ambulance ': 103, ' firefighters ': 101}
print ( phones [' police '])
```

Якщо такого ключа в словнику немає, то виникне помилка.

Видаляється елемент зі словника за допомогою спеціальної команди ***del***. Це НЕ функція, після слова ***del*** ставлять пробіл, потім пишуть ім'я словника, а далі, у квадратних дужках, ключ що видаляється:

```
phones = { ' police ': 102, ' ambulance ': 103, ' firefighters ': 101}
del phones [ ' police ']
print ( phones )
```

Перевірка приналежності ключа до словника здійснюється за допомогою операції ***key in dictionary*** (як перевірка приналежності елемента множини).

Словник є ***iterable*** і повертає ключі у випадковому порядку. Наприклад, такий код означатиме вміст словника:

```
phones = { ' police ': 102, ' ambulance ': 103, ' firefighters ': 101}
for service in phones:
    print (service, phones [ service ])
```

Приклад розв'язання складної задачі зі словниками

Розглянемо задачу. Словник заданий у вигляді набору рядків, у кожному з яких записано слово англійською мовою, потім – символ «-», далі через кому перелічені можливі переклади слова латиною.

Потрібно скласти латино-англійський словник і вивести його в тому самому вигляді. Усі слова необхідно впорядковувати за алфавітом. Можливі переклади одного слова потрібно також впорядковувати за алфавітом.

Для введення:

```
3
apple - malum, pomum, popula
fruit - baca, bacca, popum
punishment - malum, multa
```

Виведення повинно виглядати так:

```
7
baca - fruit
bacca - fruit
malum - apple, punishment
multa - punishment
pomum - apple
popula - apple
popum - fruit
```

Ідея розв'язку полягає в такому: «потрібно» розрізати кожен рядок на англійські і латинські слова. Кожне латинське слово буде ключем: додамо до його значення англійське слово (перекладів може бути декілька). Потім переглянемо відсортовані ключі і для кожного ключа виведемо відсортований список перекладів:

```
n = int ( input ())
latinEnglish = {}
for i in range (n):
    line = input ()
    english = line [: line.find ( '-')]. strip ()
    latinsStr = line [ line.find ( '-') + 1:]. strip ()
    latins = map ( lambda s: s.strip (), latinsStr.split ( ','))
    for latin in latins:
        if latin not in latinEnglish:
            latinEnglish [ latin ] = []
        latinEnglish [ latin ]. append ( english )
print ( len ( latinEnglish ))
for latin in sorted ( latinEnglish ):
    print ( latin, '-', ','. join ( sorted ( latinEnglish [ latin ])))
```

Парадигми програмування і функціональне програмування

Мови програмування пропонують різні засоби для декомпозиції задачі. Існує кілька парадигм програмування.

Імперативне (структурне, процедурне) програмування: програми –

послідовність інструкцій, які можуть читати і записувати дані з пам'яті. Під час розгляду попередніх тем використовувалася здебільшого імперативна парадигма. Деякі мови, такі як ***Паскаль*** (не ***Object Pascal***) або ***C*** є яскравими зразками імперативних мов.

Декларативне програмування: описується завдання і очікуваний результат, але не описуються шляхи його вирішення. Яскравим зразком є мова запитів до баз даних SQL: велика частина внутрішньої будови прихована в СУБД, програміст описує тільки структуру бази даних і очікуваний результат запитів.

Об'єктно-орієнтоване програмування: програми маніпулюють наборами об'єктів, до того ж об'єкти володіють можливостями і методами для змінювання стану, що зберігається в часі (або створення нових об'єктів). Прикладом мови з об'єктно-орієнтованою парадигмою є ***Java***, ***ООП*** також підтримується в ***Python*** і ***C ++***.

Функціональне програмування: завдання розбивається на набір функцій. В ідеалі, функції тільки набувають параметрів і повертають значення, не змінюючи стану об'єктів або програми. Зразком функціональних мов є ***Haskell***.

Об'єктно-орієнтоване програмування

Об'єктно-орієнтоване програмування є однією з парадигм програмування, створеній на базі імперативного програмування для зручнішого повторного використання коду й полегшення прочитаності програм.

ООП не є найоптимальнішим способом, який вирішує всі завдання. Імперативне програмування зручне для вирішення простих завдань, що використовують стандартні об'єкти. Повторне використання коду в імперативній парадигмі забезпечується за допомогою циклів і функцій, записаних в імперативному стилі.

Функціональне програмування зручне для задач з яскраво вираженим потоком даних (***data flow***), який, змінюючись, «перетікає» з однієї функції в іншу.

ООП ж забезпечує повторне використання коду за рахунок того, що

оброблювані програмою об’єкти мають багато спільного і лише незначні відмінності. ООП базується на трьох концепціях: інкапсуляція, успадкування, поліморфізм.

Інкапсуляція – це приміщення в «капсулу»: логічне об’єднання даних і функцій для роботи з ними, а також приховування внутрішньої будови об’єкта з наданням інтерфейсу взаємодії з ним (публічні методи).

Спадкування – це отримання нового типу об’єктів на основі вже існуючого, із частково або повністю запозиченою у батьківського типу функціональністю.

Поліморфізм – це надання однакових засобів взаємодії з об’єктами різного спрямування (походження). Наприклад, ***операція*** **+** може працювати як з числами, так і з рядками незважаючи на різну природу цих об’єктів.

Класом в ***Python*** називається опис структури об’єкта (полів структури) і методів (функцій) для роботи з даними в цій структурі.

Об’єктом називається екземпляр класу, де поля заповнені конкретними значеннями. Об’єкти містяться в пам’яті програми і можуть змінювати свій стан або виконувати дії за допомогою виклику методів класу для цього об’єкта.

Інкапсуляція і конструктори

Ключове слово ***class*** уже використовувалося для створення структур – набору іменованих полів, сукупність яких описує об’єкт. Однак для їх обробки застосовувались окремі функції або частини коду.

Набагато зручніше, коли опис структури об’єкта і методів роботи з ним розташовуються поруч, для зручності вивчення, модифікації і використання.

Розглянемо елементи ООП на прикладі комплексних об’єктів. Це математичний об’єкт, який складається з дійсної (***real***) і уявної (***imaginary***) частин. Запис цього числа виглядає так: `re + im*i`, де `i` – це квадратний корінь з -1.

Для створення нових об’єктів класу використовується спеціальний метод, який називається «конструктор». Методи класу записуються всередині опису класу як функції, конструктор повинен називатися **__** ***init*** **__**. Як перший

параметр у них має застосовуватися змінна ***self*** – конкретний об'єкт класу, з яким вони працюють.

Розглянемо клас для комплексного числа, виклик конструктора і друк полів об'єкта:

```
class Complex:
    def __ init __ ( self, re = 0, im = 0):
        self.re = re
        self.im = im

a = Complex (1, 2)
b = Complex (3)
c = Complex ()
print (a.re, a.im)
print (b.re, b.im)
print (c.re, c.im)
```

Конструктор містить три параметри: ***self*** – порожній об'єкт класу ***Complex***, ***re*** і ***im*** за замовчуванням, дорівнюють нулю. Виклик конструктора здійснюється за допомогою написання назви класу, у дужках вказуються параметри конструктора (усі, крім ***self***). Назви класів прийнято записувати з великої літери, а об'єкти – з маленької.

Виведенням цієї програми буде

```
1 2
3 0
0 0
```

Змінні конкретного об'єкта класу, переданого як параметр ***self***, міняються. Якщо змінні в описі класу створені, то їхні значення доступні у всіх об'єктах цього класу і їх можна змінити, перелічивши їхні імена на початку методу після виразу ***nonlocal***. Такі змінні називаються статичними, зазвичай вони призначені для зберігання якихось констант (що дуже зручно якщо потрібно, наприклад, описати якийсь клас для фізичних обчислень). Їх потрібно буде змінити, можливо, в окремих незвичайних ситуаціях, наприклад, під час підрахуванні кількості об'єктів класу.

Визначення методів та стандартні функції

Деякі стандартні функції мови ***Python*** є всього лише обгортками щодо виклику методу для переданого параметра. Наприклад, функція ***str*** викликає

метод **__ *str* __** для свого параметра. Якщо описано такий метод для цього класу, то можна застосовувати до нього функцію ***str*** явно і неявно (наприклад, вона автоматично викликається при виклику ***print*** для об'єкта цього класу).

Потрібно, щоб **__ *str* __** повертав текстове подання комплексного числа. Наприклад, число з дійсною частиною ***1*** і уявною ***2*** потрібно подати у вигляді рядка "***1+2i***", а число з дійсною частиною ***3*** і уявною ***-4.5***, як "***3-4.5i***". Повний опис класу з доданим методом буде виглядати так:

```
class Complex:
    def __ init __ ( self, re = 0, im = 0):
        self.re = re
        self.im = im
    def __ str __ ( self ):
        strRep = str (self.re)
        if self.im> = 0:
            strRep + = '+'
        strRep + = str (self.im) + 'i'
        return strRep

a = Complex (1, 2)
print (a)
b = Complex (3, -4.5)
print (b)
```

Перевизначення операторів

У мові ***Python*** можна перевизначити й поведінку операторів. Наприклад, якщо у нас є два числа x і y, то запис ***x + y*** перетворюється на виклик методу ***x.__ add __ (y).*** Значок операції + є лише зручним перевизначенням виклику методу ***add***.

Для комплексних чисел логічно визначена операція додавання: це складання окремо дійсних і окремо уявних частин. Унаслідок виклику методу для складання двох чисел має конструюватися новий об'єкт класу ***Complex***, а передані в як параметри об'єкти не повинні змінюватися. Дійсно, коли виконується операція $z = x + y$ для звичайних чисел, то має сконструюватися новий об'єкт, до якого «прив'яжеться» посилання z, а x і y не зміняться.

Тієї самої послідовності дотримуються і під час складання двох комплексних чисел. Метод **__ *add* __** повинен набувати двох параметрів,

кожен з яких є комплексним числом:

```
class Complex:
    def __ init __ ( self, re = 0, im = 0):
        self.re = re
        self.im = im
    def __ str __ ( self ):
        strRep = str (self.re)
        if self.im> = 0:
            strRep + = '+'
        strRep + = str (self.im) + 'i'
        return strRep
    def __ add __ ( self, other ):
        newRe = self.re + other.re
        newIm = self.im + other.im
        return Complex ( newRe, newIm )

a = Complex (1, 2)
b = Complex (3, -4.5)
print (a + b)
```

Перевизначення методу ***add*** має сенс тільки тоді, коли для програміста, який використовує такий клас, буде очевидним сенс операції +. Наприклад, якщо створити клас для опису деяких характеристик людини, то операція + для двох об'єктів – людей буде сприйматися різними користувачами цього класу по-різному, залежно від розвиненості фантазії читача. Такого неоднозначного розуміння доцільніше уникати і не перевизначати операцію +, якщо результат її роботи не очевидний.

Перевірка класу об'єкта

Перевизначимо для комплексних чисел ще одну операцію – множення. До того ж бажано навчитися множити комплексні числа як на цілі, або дійсні, так і на інші комплексні числа.

При множенні комплексного числа виду ***a + b * i*** на ціле, або дійсне, число x результатом буде комплексне число ***a * x + b * x * b***.

Множення двох комплексних чисел здійснюється аналогічно до множення двох многочленів першого ступеня:

```
(a + b * i) * (c + d * i) = a * c + (a * d + b * c) * i + b * d * i ** 2
```

Відомо, що i ** 2 = -1. Отже, кінцевий результат множення буде

виглядати так:

```
a * c - b * d + (a * d + b * c) * i
```

Залишилося зрозуміти, як визначити, яке число передано в метод: комплексне чи некомплексне.

У мові ***Python*** існує функція ***isinstance***, яка як перший параметр приймає об'єкт, а як другий – назву класу. Вона повертає істину, якщо об'єкт належить до цього класу, і помилку – в іншому випадку. Ця функція дозволить нам досягти потрібної функціональності від методу ***__ mul __***, що множить числа:

```
class Complex:
    def __ init __ ( self, re = 0, im = 0):
        self.re = re
        self.im = im
    def __ str __ ( self ):
        strRep = str (self.re)
        if self.im> = 0:
            strRep + = '+'
        strRep + = str (self.im) + 'i'
        return strRep
    def __ add __ ( self, other ):
        newRe = self.re + other.re
        newIm = self.im + other.im
        return Complex ( newRe, newIm )
    def __ mul __ ( self, other ):
        if isinstance ( other, Complex ):
            newRe = self.re * other.re - self.im * other.im
            newIm = self.re * other.im + self.im * other.re
        elif isinstance ( other, int ) or isinstance ( other, float ):
            newRe = self.re * other
            newIm = self.im * other
        return Complex ( newRe, newIm )
    __ rmul __ = __ mul __

a = Complex (1, 2)
b = Complex (3, -4.5)
print (a * b)
print (a * 2)
```

Крім доданого методу ***__ mul __*** на увагу заслуговує також рядок ***__ rmul __ = __ mul __***. Це присвоювання одного методу (функції) іншому, тобто, при виклику методу ***__ rmul __*** буде викликатися той самий метод ***__ mul __***.

У мові ***Python*** операція $a \times b$ замінюється на виклик методу ***a.__ mul __ (b)***. Якщо ***a*** було комплексним числом, а ***b*** – дійсним, то викликається

метод ***__ mul __*** для об'єкта ***a*** класу ***Complex***.

Однак, якщо ***a*** було дійсним числом, а ***b*** – комплексним, то виникає спроба викликати метод ***__ mul __*** для об'єкта класу ***float***. Зрозуміло, розробники стандартної бібліотеки мови ***Python*** не припускали, що буде записано клас ***Complex*** для множення на нього дійсного числа, тому робота методу ***__ mul __***, де як параметр передається щось невідоме, буде помилковою. Щоб уникнути таких ситуацій у мові ***Python,*** після невдалої спроби здійснити ***a.__ mul __ (b)*** виконується дія ***b.__ rmul __ (a)***, яка буде успішною.

Обробка помилок

Час від часу в програмах виникають помилкові ситуації, які не можуть бути оброблені в тому місці, де виникла помилка, а повинні так чи інакше оброблятися у зовнішній частині програми.

Наприклад, якщо на етапі виконання проміжної логіки виявилося, що потрібно записати рядок туди, де повинно бути число, то вже нічого не можна зробити. У такому разі потрібно виходити зі стека викликів функцій або методів проміжної логіки доти, доки не дістанемось фронтенда, який повідомить користувачеві про те, що він увів неприпустиме значення і попросить, наприклад, увести його ще раз.

Можливо, виникла ситуація, коли після заповнення дуже великої форми неможливо було її відправити, і з'являлося віконце зі словом «помилка» без будь-яких уточнень. Це недоречний стиль, оскільки повідомлення про помилку повинно бути інформативним, щоб можна було швидко її знайти й виправити. Таким чином, при виникненні помилки потрібно передавати про неї вичерпну інформацію.

У прикладі з дійсними числами можна розглянути такий приклад помилки: множення комплексного числа на будь-що, відмінне від цілого, дійсного або комплексного числа. Якщо дійти до етапу множення, то вже неможливо виправити цю помилку, окрім як повідомити про неї в тому місці, з якого була викликана операція множення.

До того ж на етапі виклику операції множення можна виконати певні дії.

Наприклад, повідомити користувачеві про його помилку і попросити ввести комплексне число. Або, якщо обробляється послідовність, із якої потрібно виокремити і перемножити комплексні числа – перейти до наступного елемента послідовності. На етапі, коли виконується невдала операція множення, неможливо передбачити, як поведе себе певна програма в разі виникнення такої помилки.

Якщо дійти до помилкової операції, то можна сконструювати спеціальний клас, який містить докладний опис помилки, і відіслати його туди, де його можна обробити.

«Викидається» помилка за допомогою команди ***raise***, а «ловиться» блоком ***try-except***. У разі множення комплексного числа на помилкове значення можна сконструювати клас помилки, що містить посилання як на комплексне число, так і на другий аргумент методу множення.

Клас для помилки повинен наслідувати стандартний клас ***BaseException***. Це означає тільки те, що в разі створення опису класу помилки потрібно написати в дужках після його назви ***BaseException***. Потенційно помилкові дії мають виконуватися в блоці ***try***, а команди для обробки помилки повинні міститися в блоці ***except***. Приклад обробки помилки буде виглядати так:

```
class ComplexError ( BaseException ):
    def __ init __ ( self, Complex, other ):
        self.arg1 = Complex
        self.arg2 = other

class Complex:
    def __ init __ ( self, re = 0, im = 0):
        self.re = re
        self.im = im
    def __ str __ ( self ):
        strRep = str (self.re)
        if self.im> = 0:
            strRep + = '+'
        strRep + = str (self.im) + 'i'
        return strRep
    def __ add __ ( self, other ):
        newRe = self.re + other.re
        newIm = self.im + other.im
        return Complex ( newRe, newIm )
    def __ mul __ ( self, other ):
```

```
            if isinstance ( other, Complex ):
                newRe = self.re * other.re - self.im * other.im
                newIm = self.re * other.im + self.im * other.re
            elif isinstance ( other, int ) or isinstance ( other, float ):
                newRe = self.re * other
                newIm = self.im * other
            else:
                raise ComplexError ( self, other )
            return Complex ( newRe, newIm )
        __ rmul __ = __ mul __

a = Complex (1, 2)
try:
    res = a * ' abcd '
except ComplexError as ce:
    print ( ' Error in mul with args:', ce.arg1, ce.arg2)
```

Виведення цієї програми:

```
Error in mul with args: 1 + 2i abcd
```

За ним легко зрозуміти, що помилка виникла під час операції множення і її передано як аргумент.

Після команди ***except*** можна вказати ім'я класу помилки, який він повинен обробляти, потім написати «as» і вказати ім'я змінної, у яку потрапить об'єкт з описом певної помилки.

Для обробки помилок можна використати різні типи блоків ***except***. Перевірка виконується послідовно. Виконується той блок команд, у якому ім'я класу співпадає з іменем класу помилки або є його попередником в ієрархії успадкування.

Проєктування структури класів

Структура опису класів становить «дерево» (у ***Python*** – ациклічний граф), успадкований від єдиного «кореня» – базового порожнього класу.

За допомогою розумно спроєктованої структури класів можна домогтися легкого прочитання і максимального повторного використання коду, забезпечення єдиного інтерфейсу тощо.

Однак, внесення фічі або зміна на високому рівні ієрархії класів може спричинити необхідність виконання величезної кількості робіт щодо

модифікації всіх послідовників цього класу, а також повну несумісність із попередньою версією. Численні зміни такого спрямування спричиняють виникнення конструкцій, які неможливо зрозуміти і (налагодити) виправити.

Водночас, закладення перспективних фічей у структуру класів призводить до ускладнень і зводити нанівець усі повторні спроби використання коду внаслідок громіздкості конструкцій. Крім того, перспективні фічі можуть бути недостатньо продуманими і спричинять появу ще більших спотворень, коли потрібно буде їх реалізувати.

Таким чином, грамотне проєктування системи класів потребує не тільки ознайомлення з патернами проектування, але й значного практичного досвіду. На початковому етапі варто навчатися проєктувати системи, у які не планується вносити зміни.

3 ОРГАНІЗАЦІЯ ТА ЗМІСТ САМОСТІЙНОЇ РОБОТИ

Питання для самоперевірки складені за матеріалами всієї дисципліни «Спеціалізоване програмне забезпечення» і є для студентів допоміжним засобом вивчення пропонованого курсу. Нижче наведено питання щодо дисципліни і перелік літературних джерел, які рекомендовано для самостійного опрацювання.

Змістовий модуль 1 Основи програмування на мові *Python*

Тема 1. Вступ до програмування на мові ***Python***

Література: [1 – 14]

Перелік питань

1. Визначте поняття «програма» та «мова програмування».
2. Назвіть основні етапи розвитку мов програмування?
3. Історія створення та особливості мови програмування ***Python***.
4. Інтерпретатор і середовище програмування у ***Python***.

Тема 2 Типи даних у мові програмування ***Python***

Література: [1 – 14]

Перелік питань

1. Убудовані типи даних мови програмування ***Python***.
2. Цілочисельний тип даних мови програмування ***Python***.
3. Тип чисел з плавкою точкою мови програмування ***Python***.
4. Рядки у мові програмування ***Python***.
5. Перетворення типів даних: функції ***int(), float(), str()***.
6. Логічний тип даних мови програмування ***Python***.
7. Виведення даних з клавіатури в мові програмування ***Python***.
8. Уведення даних з клавіатури в мові програмування ***Python***.

9. Створення списків у мові програмування ***Python***.
10. Кортежі в мові програмування ***Python***.
11. Словники в мові програмування ***Python***.
12. Множини в мові програмування ***Python***.
13. Складені структури даних у мові програмування ***Python***.

Змістовий модуль 2 Особливості реалізації алгоритмів на мові програмування *Python*

Тема 3 Алгоритмічні структури у мові програмування ***Python***

Література: [1 – 14]

Перелік питань

1. Структури коду в мові програмування ***Python***.
2. Створення та перевірка умов у мові програмування ***Python***.
3. Розгалуження в мові програмування ***Python***.
4. Створення та перевірка повторень у мові програмування ***Python***.
5. Включення (скорочення синтаксису) у мові програмування ***Python***.
6. Генератори у мові програмування ***Python***.
7. Функції у мові програмування ***Python***. Виняткові ситуації під час обробки помилок.

Тема 4 Модулі та пакети у мові програмування ***Python***

Література: [1 – 14]

Перелік питань

1. Імпорт модулів: інструкція import у мові програмування ***Python***.
2. Пакети у мові програмування ***Python***.
3. Стандартна бібліотека мови програмування ***Python***.

Змістовий модуль 3 Розробка додатків з використанням мови програмування *Python*

Тема 5 Об’єктно-орієнтований підхід у мові програмування ***Python***

Література: [1 – 14]

Перелік питань

1. Особливості об’єктно-орієнтованого програмування та об’єкти мови програмування ***Python***.
2. Створення та використання класу в мові програмування ***Python***.
3. Створення класів та екземплярів на мові програмування ***Python***.
4. Наслідування в мові програмування ***Python***.
5. Перевизначення методу в мові програмування ***Python***.
6. Екземпляри, як атрибути в мові програмування ***Python***.

Тема 6 Програмування додатків баз даних на мові програмування ***Python***

Література: [1 – 14]

Перелік питань

1. Групи файлів та особливості роботи з ними.
2. Бази даних у мові програмування ***Python***.
3. Особливості роботи із датою та часом у мові програмування ***Python***.

4 РОЗРАХУНКОВО-ГРАФІЧНЕ ЗАВДАННЯ

Як індивідуальне завдання для денної та заочної форм навчання виконується розрахунково-графічна робота на тему: «Використання спеціалізованих модулів та додатків геоінформаційних технологій щодо пошуку інвестиційно-привабливої території» – 30 год.

Мета: дослідити можливості та особливості практичного використання геоінформаційних технологій для пошуку інвестиційно-привабливої території та візуалізації її у тривимірному форматі.

Вступ

Процес інвестування в місті має свої особливості, пов’язані зі специфікою джерел інвестування, сукупністю їхніх форм та обмеженістю стадій відтворення в міських структурах. Саме тому до компетенцій міських органів влади належать важелі, які можуть істотно покращити інвестиційний клімат міста. Привабливість та інвестиційний клімат міста значною мірою визначаються і забезпечуються сформованою міською адміністрацією інвестиційною політикою, яка повинна містити комплекс заходів підтримки інвестиційної діяльності та структурних перетворень у виробничій і соціальній сферах, забезпечувати зростання обсягів промислового виробництва.

Величезний обсяг різноманітної і дорогої інформації, отриманої різними методами з різних джерел, необхідно використовувати ефективно. Для цього, насамперед, потрібно подавати цю інформацію в наочній і доступній для огляду формі, що дозволяє швидко виокремлювати найбільш важливі її складники для подальшого аналізу та прийняття обґрунтованих рішень.

Інвестиційно-привабливі земельні ділянки щонайкраще відображаються на географічній карті міста і його області. Отже, найбільш наочне й узагальнююче подання та аналіз зазначеної інформації можуть бути виконані тільки за допомогою сучасних геоінформаційних систем.

Опис предметної та проблемної області досліджень

Виконується обґрунтування актуальності обраної теми.

Об’єктом вивчення у розрахунково-графічній роботі є земельна ділянка території населеного пункту, предмет якої – інвестиційно-приваблива територія.

Метою роботи є дослідження можливостей та особливостей практичного використання геоінформаційних технологій для візуалізації інвестиційно-привабливої території у тривимірному форматі.

Завданням роботи є створення моделі поліпшення інвестиційно-привабливої території населеного пункту та її тривимірної моделі за допомогою геоінформаційних технологій; розробка модулів та додатків на мові програмування ***Python***.

У розрахунково-графічній роботі обґрунтовується доцільність будівництва на певній території обраного об’єкта інфраструктури (торгового центру, заводу, закладу освіти тощо), подаються його характеристики та корисні властивості, враховується наявність аналогових будівель.

Використання території, на якій планується будівництво обраного об’єкта, повинно бути привабливою пропозицією для населеного пункту, що дасть змогу місту стати більш привабливим для міських мешканців, забезпечити робочі місця та перспективи для розвитку бізнесу, дозволить поповнити кошти міського бюджету тощо.

Характеристика вихідних даних досліджуваної проблеми

Цей розділ розрахунково-графічної роботи містить характеристики вибраної земельної ділянки. Зазначається адреса розташування ділянки, її площа, наявність будівель, споруд, їхні площа, стан та призначення.

Вибір інструментального засобу для вирішення визначеного завдання

Одним з найважливіших питань, яке розглядається під час вивчення й порівнянні різних інформаційних систем, зокрема геоінформаційних систем (ГІС), є технологія зберігання й роботи з даними. Це пов'язано з тим, що найбільш цінним компонентом системи є дані, а не програми або обладнання, за якими ця система працює. До того ж у процесі експлуатації системи вартість програм і обладнання зменшується, внаслідок старіння або зношування, тоді як цінність даних весь час збільшується. Розглядаються основні технологічні схеми ГІС з погляду внутрішньої організації роботи й моделі зберігання просторових даних.

Перша технологічна схема побудови ГІС – одна або кілька програм, об'єднаних у програмну систему, що запускаються на комп'ютері користувача. Для зберігання використовується внутрішній формат даних, здебільшого закритий для використання правовласником (обмеження в ліцензійній угоді, наявність патентів тощо).

Друга технологічна схема побудови ГІС базується на технології клієнт-сервер для організації роботи з даними в комп'ютерній мережі, має програму-клієнта для кінцевого користувача і програму-сервер, яка веде базу просторових даних. До того ж використовується власна структура бази даних і внутрішні формати даних, зазвичай захищені авторськими правами. Багато систем є подальшим розвитком ГІС першої технологічної схеми для організації роботи в комп'ютерній мережі, тому працюють із файлами даних тих самих форматів.

Третя технологічна схема побудови ГІС – додаток для кінцевого користувача або система побудована за схемою клієнт-сервер, які для зберігання просторових даних використовують одну з поширених систем управління базами даних (СУБД), останнім часом здебільшого на базі одного з поширених ***SQL-серверів*** (***Microsoft SQL Server, Oracle, MySQL, PostgreSQL***). До того ж внутрішня структура зберігання просторових даних є унікальною для цієї ГІС, зазвичай закритою для використання правовласником.

Четверта технологічна схема побудови ГІС базується на використанні як сховища просторових даних спеціалізованих розширень для найбільш поширених ***SQL-серверів***, які на сьогодні мають усі основні постачальники подібних рішень.

Ще однією тенденцією, характерною для вирішення такого покоління ГІС, є перехід до використання як робочого місця кінцевого користувача ГІС додатків на основі WEB-браузера.

Розглядаються платформи для побудови ГІС.

Геоінформаційне моделювання території населеного пункту

Основні принципи геоінформаційного моделювання. Цей розділ містить теоретичні відомості щодо основних принципів геоінформаційного моделювання.

Геоінформаційне моделювання забезпечує формалізоване уявлення використовуваних даних і їхніх взаємозв'язків, тому сучасне вміння працювати з інформацією означає вміння здійснювати геоінформаційне моделювання.

Таким чином, геоінформаційне моделювання можна розглядати, як сучасну інформаційну технологію. Воно включає вміння створювати різні інформаційні моделі, їх інтерпретувати й застосовувати.

Геоінформаційна модель містить кілька рівнів опису:

– предметний, пов'язаний з областю обробки інформації;

– системний, пов'язаний з методами організації та способами обробки;

– базовий, який визначається вибором базових моделей даних, незалежних від області застосування інформаційної моделі.

Переваги подання території у вигляді тривимірної моделі. Застосування тривимірних моделей дозволяє наочно зображувати (візуалізувати) об'єми і розв'язувати задачі, пов'язані з їхнім моделюванням, по-новому розв'язувати задачі проєктування житлової забудови, розміщення об'єктів побутового й господарського призначення в муніципальних системах, створювати нові типи тривимірних умовних знаків, розширювати можливості

ГІС, як системи підтримки і прийняття рішень, виконувати синтез тривимірних структур тощо.

Пошук інвестиційно-привабливої території населеного пункту

Критерії пошуку інвестиційно-привабливої території населеного пункту. До основних критеріїв інвестиційної привабливості території належать:

– площа території;

– ресурсний потенціал (баланси енергетики, газопроводу, трубопроводу, земельних ресурсів, природних ресурсів);

– потенціал інфраструктури;

– трудовий потенціал;

– близькість до аналогічних об’єктів;

– транспортна доступність;

– рівень економічного розвитку;

– стан довкілля;

– рівень загальної безпеки;

– активність території щодо інвесторів, інвестиційна діяльність у регіоні.

Для наочності у таблиці 5 подається порівняльна оцінка кожної території за критеріями. Критерії переведені у п’ятибальну шкалу, за якою здійснювалася оцінка критеріїв по кожній території.

Таблиця 5 – Порівняльна оцінка територій за критеріями

Критерій	Обрана територія	Аналог 1	Аналог 2	Аналог 3	Аналог 4
Площа території, га	41,7 га	14,5 га	21,6 га	4,4 га	2 га
Ресурсний потенціал (баланси енергетики, газопроводу, трубопроводу, земельних ресурсів, природних ресурсів)	5	0	2	2	0
Потенціал інфраструктури	5	0	1	2	0

Продовження таблиці 5

Трудовий потенціал	5	3	1	4	3
Близькість до аналогічних об'єктів	4	1	2	4	5
Транспортна доступність	5	3	2	4	3
Рівень економічного розвитку	5	2	1	4	3
Стан довкілля	5	3	5	4	3
Рівень загальної безпеки	4	2	2	4	5
Активність території щодо інвесторів, інвестиційна діяльність у регіоні	0	0	1	1	0

Вибір інвестиційно-привабливої території населеного пункту. Згідно з таблицею 5 територія для побудови обирається за вищими балами критеріїв.

Опис запропонованої моделі поліпшення інвестиційно-привабливої території

Розділ розрахунково-графічної роботи містить опис об'єкта на обраній території. Зазначаються призначення вибраного об'єкта, місце розташування, площа будівництва, переваги такого об'єкта для населеного пункту. Подається план будівлі.

Створення необхідної бази даних та векторизація запропонованої моделі поліпшення інвестиційно-привабливої території міста

Створюється необхідна база даних для запропонованої моделі поліпшення інвестиційно-привабливої території населеного пункту. Додаток ***ArcCatalog*** використовується для організації, роботи та управління географічною інформацією в робочих областях і базах геоданих.

Бази геоданих – це зібрання наборів географічних даних різних типів, які використовуються в ArcGIS.

ArcCatalog подає ці дані у вигляді «деревоподібної структури каталогу», чим полегшує роботу з ними. Він є своєрідним аналогом ***Провідника Windows***, призначеним для роботи з документами і наборами даних ***ArcGIS***.

Запускаємо ***ArcCatalog***. Обговорюємо місце розташування ***БГД***. За допомогою контекстного меню в папці, у якій буде розташована ***БГД,*** обираємо необхідні команди (рис. 2).

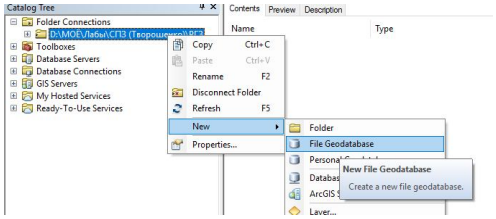

Рисунок 2 – Створення ***БГД*** через контекстне меню ***ArcCatalog***

Після створення необхідної бази робота продовжується в ***ArcMap***.

Векторизація запропонованої моделі поліпшення інвестиційно-привабливої території міста. Створену базу даних завантажуємо в ***ArcMap***. На обраній території необхідно зобразити майбутню будівлю, тому потрібно пов’язати плани поверхів з місцем забудови.

Інтеграція створення шарів запропонованої моделі в єдину тривимірну модель за допомогою геоінформаційних технологій

Після того як були створені просторові об’єкти та внесена вся необхідна атрибутивна інформація, відбувається процес інтеграції поверхових шарів в єдину модель за допомогою додаткового модуля ***ArcScene***.

Розробка спеціалізованих модулів та додатків з використанням мови програмування *Python*

Програмування спеціалізованих додатків бази даних на мові програмування Python

Задача № 1. Розрахувати загальну вартість оренди приміщень, якщо приміщення першого поверху мають коефіцієнт 1,4, другого поверху – 1,2, а третього поверху – 1, враховуючи вартість 1 м^2.

Задача № 2. Розрахувати кількість ламінату (м^2) для першого та другого поверхів та їхню вартість, враховуючи вартість квадратного метра ламінату, а

також кількість лінолеуму ($м^2$) для третього поверху та його вартість, враховуючи вартість квадратного метра лінолеуму.

Задача № 3. Створити словник, який містить номер приміщення та його площу по всій будівлі.

Задача № 4. Розрахувати кількість трьох літрових банок фарби та їхню вартість, врахувавши норму однієї трьох літрової банки на квадратний метр та її вартість, для фарбування фасаду будівлі.

Задача № 5. Розрахувати кількість паркувальних місць, якщо норма на одне місце – 23 $м^2$ та вільну територію – 168 832,4 $м^2$.

Задача № 6. Розрахувати кількість ліхтарів та лавок, враховуючи норми ДБН, а також довжину лавки та довжину доріжок.

Висновки

У розрахунково-графічній роботі досліджено можливості та особливості практичного використання геоінформаційних технологій для пошуку інвестиційно-привабливої території та візуалізації її у тривимірному форматі.

За допомогою геоінформаційних технологій створено модель об’єкта інфраструктури на інвестиційно-привабливій території населеного пункту та її тривимірну модель, розроблено модулі та додатки на мові програмування ***Python***.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Творошенко І. С. Спеціалізоване програмне забезпечення : конспект лекцій для магістрів денної та заочної форм навчання спеціальності 193 – Геодезія та землеустрій освітньої програми «Геодезія та землеустрій» / І. С. Творошенко ; Харків. нац. ун-т міськ. госп-ва ім. О. М. Бекетова. – Харків : ХНУМГ ім. О. М. Бекетова, 2018. – 118 с.
2. Сузи Р. А. Python. Наиболее полное руководство / Р. А. Сузи. – СПб. : БХВ-Петербург, 2002. – 768 с.
3. Шипулін В. Д. Основні принципи геоінформаційних систем / В. Д. Шипулін. – Харків : ХНАМГ, 2012. – 312 с.
4. Лутц М. Изучаем Python / М. Лутц. – СПб. : Символ-Плюс, 2011. – 1280 с.
5. Лутц М. Программирование на Python : в 2 томах / М. Лутц. – СПб. : Символ-Плюс, 2011. – Т. 1. – 992 с.
6. Дэвид М. Бизли Python. Подробный справочник / Дэвид М. Бизли. – СПб. : Символ-Плюс, 2010. – 864 с.
7. Саммерфилд М. Программирование на Python 3. Подробное руководство / М. Саммерфилд. – СПб. : Символ-Плюс, 2009. – 608 с.
8. Саммерфилд М. Python на практике / М. Саммерфилд – М. : ДМК Пресс, 2014. – 338 с.
9. Сузи Р. А. Язык программирования Python : учеб. пособие / Р. А. Сузи. – М. : ИНТУИТ, БИНОМ. Лаборатория знаний, 2006. – 328 с.
10. Доусон М. Программируем на Python / М. Доусон. – СПб. : Питер, 2012. – 432 с.
11. Дэвид М. Бизли Язык программирования Python. Справочник / Дэвид М. Бизли. – Киев : ДиаСофт, 2000. – 336 с.
12. Хахаев И. А. Практикум по алгоритмизации и программированию на Python : учебник / И. А. Хахаев. – М. : Альт Линукс, 2010. – 126 с.
13. Методичні вказівки до вивчення дисципліни «Основи геоінформатики» для студентів освітніх напрямів: 6.090106 – Екологія, охорона

навколишнього середовища та збалансоване природокористування; 6.090103 – Лісове і садово-паркове господарство; 6.090101 – Агрономія, спеціальність 8.09010104 – Плодівництво і виноградарство ; уклад. С. П. Сонько, Ю. Ю. Косенко. – Умань, УНУС, 2013. – 103 с.

14. Мова програмування Python для інженерів і науковців: навчальний посібник [Електронний ресурс]. – Ivano-Frankivsk, Ukraine : ІФНТУНГ, 2019. – 275 с. – Режим доступу: https://vkopey.github.io/Python-for-engineers-and-scientists/. – Назва з екрана.

Виробничо-практичне видання

Методичні рекомендації
до проведення практичних,
самостійної та розрахунково-графічної робіт
з навчальної дисципліни

«СПЕЦІАЛІЗОВАНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ»

(для студентів другого (магістерського) рівня вищої освіти денної та заочної форм навчання зі спеціальності 193 – Геодезія та землеустрій)

Укладачі: **НЕСТЕРЕНКО** Сергій Григорович
МИРОНЕНКО Марія Леонідівна
КАСЬЯНОВ Володимир Володимирович
ЄВДОКІМОВ Андрій Анатолійович

Відповідальний за випуск *О. Є. Поморцева*
Редактор *О. А. Норик*
Комп'ютерне верстання *М. Л. Мироненко*

План 2021 , поз. 26 М

Підп. до друку 28.01.2021 Формат 60 х 84 / 16
Електронне видання. Ум. друк. арк. 4,2.

Видавець і виготовлювач:
Харківський національний університет
міського господарства імені О. М. Бекетова,
вул. Маршала Бажанова, 17, Харків, 61002
Електронна адреса: office@kname.edu. ua
Свідоцтво суб'єкта видавничої справи:
ДК № 5328 від 11.04.2017.