

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ  
ІМЕНІ ТАРАСА ШЕВЧЕНКА

**О. В. Обвінцев**

# **Командна робота на заняттях з програмування**

**Методичні рекомендації для викладачів**

**КИЇВ – 2021**

## Зміст

Передмова .....	3
Організація посібника .....	5
Групове завдання 1. Розрахувати кількість матеріалу для спорудження будинку та зобразити цей будинок. ....	5
Групове завдання 2. Знайти різницю площ фігур, на які пряма ділить прямокутник, та зобразити цей прямокутник та відрізок прямої. ....	7
Групове завдання 3. Стрільба з гармати. ....	9
Групове завдання 4. Знайти задану цифру послідовності. ....	10
Групове завдання 5. Максимальний паліндром .....	12
Групове завдання 6. Зображення правильних многокутників .....	13
Групове завдання 7. Гра у відгадування слова .....	15
Групове завдання 8. Побудова магічних квадратів .....	16
Групове завдання 9. Перевірка, чи є текст віршом .....	17
Групове завдання 10. Аналіз шахової позиції .....	18
Групове завдання 11. Перевірка, чи є рядок правильним виразом .....	21
Групове завдання 12. Пошук виходу з лабіринту .....	22
Групове завдання 13. Дешифрування повідомлення .....	24
Групове завдання 14. Цистеріанські числа .....	27
Групове завдання 15. Інтерпретатор лінійних програм. Крок 1 .....	33
Групове завдання 16. Інтерпретатор лінійних програм. Крок 2 .....	54
Групове завдання 17. Інтерпретатор лінійних програм. Крок 3 .....	73
Групове завдання 18. Клавіатурний тренажер .....	89
Групове завдання 19. Складання кросвордів. Крок 1 .....	93
Групове завдання 20. Складання кросвордів. Крок 2 .....	106
Групове завдання 21. Розрахунок матеріалів для будинку .....	112
Групове завдання 22. Ланцюговий код. Крок 1 .....	114
Групове завдання 23. Ланцюговий код. Крок 2 .....	118
Групове завдання 24. Фільтрація та відновлення рядків .....	123
Групове завдання 25. Обмін повідомленнями між програмами .....	125
Групове завдання 26. Введення даних форми .....	130
Список літератури .....	137

## Передмова

Командна робота у програмуванні є загально визнаним методом роботи та отримання результатів. Парне програмування, сумісне володіння програмним кодом використовуються у розповсюджених сучасних методологіях розробки програм. Однак командна робота, як правило, не знаходить свого місця у проведенні занять з навчання програмуванню та сумісним дисциплінам. Даний посібник призначений для опису досвіду такої роботи саме на заняттях з програмування, а також містить рекомендації та готові варіанти завдань для річного курсу з програмування. Джерелом слугує досвід автора щодо пропонування студентам командної роботи у курсі програмування (формально – курси «Програмування», «Об'єктно-орієнтоване програмування» для студентів 1 курсу механіко-математичного факультету, освітня програма Комп'ютерна математика) на основі мови програмування Python. Матеріали лекцій курсу викладено у [1]. Вони також доступні у інтернет за адресою <http://matfiz.univ.kiev.ua/pages/13> (на час написання даного посібника).

Треба зазначити, що командна робота не виключає та не замінює індивідуальної роботи студентів над розробкою програм, а радше слугує доповненням такої індивідуальної роботи. Також не треба плутати командну роботу з розміщенням декількох студентів під час аудиторних занять за одним комп'ютером, що було розповсюджено у часи нестачі комп'ютерної техніки і що ніколи не підтримував автор.

Командна робота на заняттях з програмування у чомусь наслідуює відому сучасну практику хакатонів, коли розробники збираються у команди та вирішують поставлені кимось або самими розробниками задачі за обмежений час. Як правило цей час сягає доби або 2 діб під час вихідних. Звичайно, для проведення занять такий термін не є прийнятним. Тому кожне командне завдання обмежено у часі однією академічною парою.

Для проведення занять у форматі командної роботи необхідно дотримання декількох вимог:

1. Технічні передумови мають надати можливість кожному студенту працювати за своїм комп'ютером (або власним ноутбуком, або комп'ютером у комп'ютерному класі). Крім того, комп'ютери мають бути об'єднані у мережу, щоб студенти у команді могли вільно обмінюватись розробленим програмним кодом та консолідувати його.
2. Організаційні передумови – це створення команд з визначеною кількістю учасників. Автор з декількох причин пропонує створювати команди з 3 учасників. Це забезпечує нескладний обмін досвідом, надає можливість активної участі усіх членів команди у роботі а також не вимагає додаткових організаційних зусиль для координації роботи. Студенти об'єднуються у команди за власним бажанням. Якщо кількість студентів не ділиться на 3, то можуть бути утворені команди за 2 учасників, для яких завдання може бути спрощене.
3. Усі команди мають отримувати однакоє завдання з метою одержання порівнюваних результатів роботи. Виконання завдання має винагороджуватись однаково для всіх учасників команди незалежно від видимої участі у досягненні результату.

Роль викладача на таких заняттях, окрім підготовки самих завдань, - це роль модератора, який має підводити команди до отримання результату, підказувати шляхи організації роботи, надавати підказки у потрібний час, якщо група погано сприймає завдання, відповідати на запитання студентів, приймати та перевіряти виконані завдання.

Автор також проводив розмежування між першою командою, що виконала завдання, та іншими командами. Перша команда отримувала більше балів за завдання, ніж інші. Це дозволяє додати елемент змагальності у роботу студентів та спонукає їх до більш відповідальної та активної роботи. За спостереженнями автора, під час таких занять майже ніхто зі студентів на перерву не виходить, натомість використовуючи цей час для отримання результату.

Командна робота має декілька додаткових плюсів. Усі учасники команди навчаються у процесі реальної роботи. При цьому відбувається активний та насичений обмін досвідом, студенти реально засвоюють тематику, що розглядається на заняттях. Студенти, у яких є труднощі з засвоєнням матеріалу, бачать, що проблеми можна вирішувати і їх вирішують такі ж студенти. Для студентів, які мають хороші результати з курсу, – це можливість підтвердити ці результати або зрозуміти, що є ще простір для додаткової роботи, і можливо, їх колеги досягли не гірших результатів. У цій роботі немає місця для списування, представлення чужих результатів як своїх (за весь час проведення був тільки один такий випадок, але це виключення, яке підтверджує правило). Уся робота відбувається у однакових умовах, що дає можливість студентам порівнювати свої знання та вміння, отже породжує бажання вдосконалювати і свої навички, і навички роботи у команді. До того ж, командна робота на заняттях яскраво демонструє риси командної роботи у реальному виробництві: учасники команд, які не демонструють бажання працювати на команди, надалі не запрошуються у ці команди.

Звичайно, підготовка завдань для командної роботи вимагає від викладача значних витрат свого часу. Це спричинено специфічними вимогами для таких завдань:

- Завдання мають бути новими, невідомими для студентів, тому їх публікація у електронних джерелах має бути обмеженою, або кількість таких завдань до кожної теми має бути значною.
- Завдання мають бути по силах студентам, тобто добре підготовлена команда повинна встигнути виконати завдання протягом пари. Якщо завдання є заскладним, - це демотивує студентів.
- Завдання не має бути дуже простим, таким, яке виконується за 10-15 хвилин, що також демотивує студентів.
- Завдання не має містити складних алгоритмічних задач, оскільки їх розробка у стислий термін суттєво залежить від способу мислення студентів, і ті студенти, які знаходять розв'язок цих завдань швидко, - будуть мати перевагу, що не є правильним з точки зору мети командної роботи. Якщо у завданні пропонуються якісь складні алгоритми, - вони мають бути пояснені або мають бути посилання на джерела з описом таких алгоритмів.
- Бажано, щоб завдання містило додатковий матеріал, який не розглядається у курсі, або розглядається у курсі пізніше (у прикладах далі – це turtle, випадкові величини). Це дає змогу урізноманітнити завдання, зробити їх цікавими для всіх.
- Бажано, щоб завдання включало проміжні результати та головний підсумковий результат. Це дозволяє ставити задачі досягнення проміжних результатів для команд, яким важко досягти підсумкового результату.
- У ряді випадків викладач має сам написати та налагодити програми для розв'язання завдань, що пропонуються. Це, зокрема, дає можливість виміряти час, який потрібен студентам для виконання завдання (звичайно, треба мати на увазі різницю у підготовці

студентів та викладача і час виконання завдання викладачем має бути набагато менше, ніж 1.5 години).

## Організація посібника

У посібнику надано завдання для командної роботи (або групові завдання).

Для завдань наведено:

- Текст завдання. Роздається у друкованому вигляді по 1 примірнику усім командам на початку заняття.
- Можливі додаткові матеріали до завдання. Розсилаються студентам на електронні адреси за добу або безпосередньо перед початком заняття. Якщо матеріали містять дані, які можуть розкрити тему завдання, вони закриваються паролем, який повідомляється студентам на початку заняття.
- Необхідні передумови – теми курсу, які мають засвоїти студенти на момент отримання завдання
- Мета завдання
- Обмеження на виконання завдання (особливо актуально для початкових тем курсу)
- Пропозиції щодо розпаралелювання роботи над завданням у команді
- Зауваження щодо вирішення завдання
- Рекомендації щодо перевірки результатів
- Тексти програм із розв'язками (для деяких завдань)
- Тексти програм для створення завдань (для деяких завдань)

Кілька завдань мають продовження, тобто пропонуються на декількох підряд заняттях. У таких випадках наведено додаткові умови щодо організації виконання завдань.

Усі завдання у форматі .pdf та наведені у посібнику програми, інші додаткові матеріали знаходяться у репозиторії git за адресою: <https://github.com/aobvintsev/teamwork>.

## Групове завдання 1. Розрахувати кількість матеріалу для спорудження будинку та зобразити цей будинок.

### Текст завдання

Будується одноповерховий будинок прямокутної форми з двоскатним дахом. Матеріал фундаменту – бетон. Матеріал стін – газоблок. Матеріал даху металочерепиця. Розміри будинку – від 7 до 15 м довжина, від 5 м до 10 м ширина, від 2,5 до 3,5 метрів висота. На кожній з довгих стін розташовано по 2 вікна симетрично від кутів, на кожній з коротших – по 1 вікну посередині. На одній з довгих стін посередині розташовано двері.

Розміри всіх вікон однакові: 1.2 x 1.6 м

Розмір дверей: 0.9 x 2 м

Дах виходить за стіну з кожного боку на 0.5 м

Розмір газоблоку: 0.6 x 0.4 x 0.2 м, товщина стін – 0.4 м

Робоча ширина листа металочерепиці 1.05 м

Ширина фундаменту 0.4 м

Задаються зовнішня довжина та ширина будинку, висота стін (з урахуванням обмежень, вказаних вище) а також висота верхньої точки даху над стіною. Також задається глибина фундаменту від рівня землі та висота фундаменту над землею.

Треба скласти програму у Python для розрахунку:

- Об'єму бетону
- Кількості газоблоків (з урахуванням допуску додатково 10% від мінімально необхідної кількості)
- Кількості та довжини листів черепиці (черепиця відрізається потрібного розміру по довжині)

Для обчислення кореня квадратного з числа, потрібно на початку програми вказати

```
from math import sqrt
```

Далі у програмі обчислення квадратного кореня з  $x$  позначається  $\text{sqrt}(x)$

Також треба зобразити проекцію будинку з одної сторони у вибраному масштабі.

Для зображення будинку використати графічну бібліотеку turtle. Для її використання треба на початку програми написати

```
import turtle
```

turtle надає графічний курсор для відтворення простих зображень у графічному режимі. Мінімумально потрібні дії з turtle вказано у таблиці

<code>turtle.up()</code>	Підняти пензель догори (припинити малювання)
<code>turtle.down()</code>	Опустити пензель донизу (почати малювання)
<code>turtle.setpos(x, y)</code>	Встановити курсор у позицію (x, y)

Наприклад, щоб зобразити лінію від точки  $(x_1, y_1)$  до точки  $(x_2, y_2)$ , треба написати послідовність команд:

```
turtle.up()
```

```
turtle.setpos(x1, y1)
```

```
turtle.down()
```

```
turtle.setpos(x2, y2)
```

### Необхідні передумови

Засвоєння Теми 1 «Лінійні програми»

### Мета завдання

Навчитись писати лінійні програми, що містять ведення з клавіатури, присвоєння та виведення. Також навчитись писати прості лінійні програми для графічного зображення простих геометричних фігур з використанням turtle.

### Обмеження на виконання завдання

Не використовувати розгалуження, цикли, функції.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Розділити обчислення та графічне відображення, потім об'єднати результати.

### Зауваження щодо вирішення завдання

Треба дотримуватись вказаних у тексті завдання обмежень при введенні даних з клавіатури. Перевірки введених даних не передбачені.

Для графічного відображення визначити початкові координати  $x\_start$ ,  $y\_start$  (наприклад, лівого нижнього кута) та масштабний коефіцієнт відображення 1 метру у  $N$  пікселях. Використовувати цей коефіцієнт у всіх діях з turtle.

## Групове завдання 2. Знайти різницю площ фігур, на які пряма ділить прямокутник, та зобразити цей прямокутник та відрізок прямої.

### Текст завдання

Скласти програму для обчислення різниці площ фігур на які пряма  $y=ax+b$  ділить прямокутник  $P=\{(x, y): a1 \leq x \leq a2, b1 \leq y \leq b2\}$ .

Розглянути усі можливі випадки взаємного розташування прямої та прямокутника. Вважати, що якщо пряма не перетинає прямокутник, площа однієї з фігур дорівнює нулю, отже різниця площ – це площа всього прямокутника.

Зобразити прямокутник та пряму (відрізок прямої) за допомогою бібліотеки turtle.

Вибрати масштаб та границі відрізка прямої так, щоб відрізок на рисунку перетинав прямокутник, якщо пряма його перетинає.

Виконати програму та показати зображення для усіх можливих різних випадків взаємного розташування прямої та прямокутника.

Для використання turtle треба на початку програми написати

```
import turtle
```

turtle надає графічний курсор для відтворення простих зображень у графічному режимі. Мінімально потрібні дії з turtle вказано у таблиці

<code>turtle.up()</code>	Підняти пензель догори (припинити малювання)
--------------------------	--

<code>turtle.down()</code>	Опустити пензель донизу (почати малювання)
<code>turtle.setpos(x, y)</code>	Встановити курсор у позицію (x, y)

Наприклад, щоб зобразити лінію від точки (x1, y1) до точки (x2, y2), треба написати послідовність команд:

`turtle.up()`

`turtle.setpos(x1, y1)`

`turtle.down()`

`turtle.setpos(x2, y2)`

### Необхідні передумови

Засвоєння Теми 2 «Розгалужені програми»

### Мета завдання

Навчитись писати розгалужені програми, що містять ведення з клавіатури, присвоєння та виведення а також розгалуження. Також навчитись писати розгалужені програми для графічного зображення простих геометричних фігур з використанням turtle.

### Обмеження на виконання завдання

Не використовувати цикли, функції.

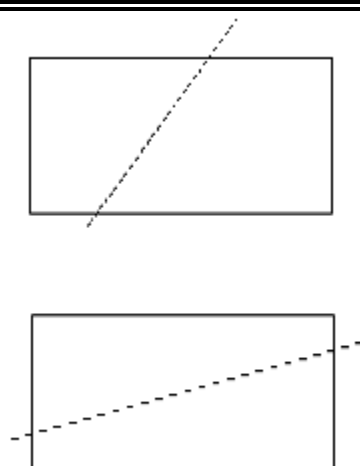
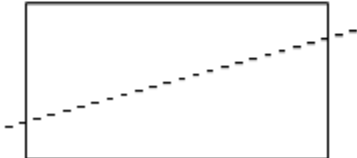
### Пропозиції щодо розпаралелювання роботи над завданням у команді

Розділити обчислення та графічне відображення, потім об'єднати результати.

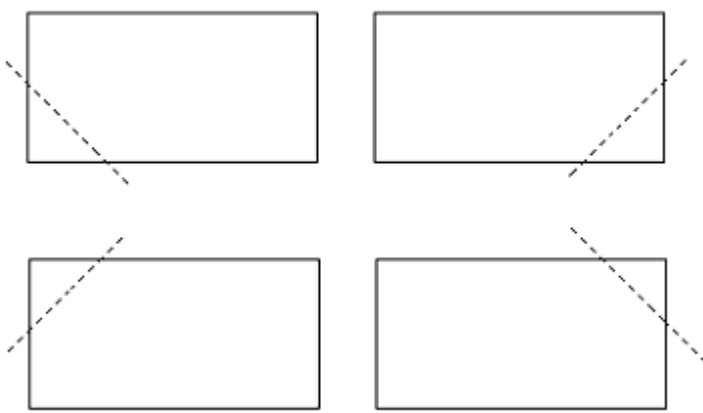
### Зауваження щодо вирішення завдання

Для розв'язку задачі розглянути 7 можливих різних випадків перетину прямокутника прямою:

$a = 0$ ,  $b_1 \leq b \leq b_2$  а також

Дві трапеції	
$a_1 \leq \frac{b_2 - b}{a} \leq a_2 \quad 3$ $a_1 \leq \frac{b_1 - b}{a} \leq a_2$ <hr/> $b_1 \leq aa_1 + b \leq b_2 \quad 3$ $b_1 \leq aa_2 + b \leq b_2$	 



Трикутник і п'ятикутник	
	$b_1 \leq aa_1 + b \leq b_2 \mid b_1 \leq aa_2 + b \leq b_2$
$a_1 \leq \frac{b_1 - b}{a} \leq a_2$ <hr/> $a_1 \leq \frac{b_2 - b}{a} \leq a_2$	

Для графічного відображення визначити початкові координати  $x\_start$ ,  $y\_start$  (наприклад, лівого нижнього кута) та масштабний коефіцієнт відображення 1 метру у  $N$  пікселях. Використовувати цей коефіцієнт у всіх діях з turtle.

### Групове завдання 3. Стрільба з гармати.

#### Текст завдання

Знайти точку, у яку влучить снаряд, якщо його випущено з гармати під кутом  $\alpha$ , зі швидкістю  $v$ . При цьому, сама гармата розташована на висоті  $h$  над землею. Показати графічно траєкторію польоту снаряду. Траєкторію показувати у turtle маленькими колами.

Для зображення кола використати функцію `turtle.circle()`

`turtle.circle(r)`, де  $r$  – радіус кола

Щоб використати у Python функції `sin`, `cos`

треба написати

```
from math import sin, cos
```

Розв'язати також таку задачу: дано точку, у яку повинен влучити снаряд. Розрахувати початкову швидкість та кут

Виконати задачі з різними значенням параметрів

Для використання turtle треба на початку програми написати

```
import turtle
```

turtle надає графічний курсор для відтворення простих зображень у графічному режимі.

Мінімально потрібні дії з turtle вказано у таблиці

<code>turtle.up()</code>	Підняти пензель догори (припинити малювання)
<code>turtle.down()</code>	Опустити пензель донизу (почати малювання)
<code>turtle.setpos(x, y)</code>	Встановити курсор у позицію (x, y)

Наприклад, щоб зобразити коло радіусу  $r$  з центром у точці  $(x_1, y_1)$ , треба написати послідовність команд:

```
turtle.up()
```

```
turtle.setpos(x1, y1 - r)
```

```
turtle.down()
```

```
turtle.circle(r)
```

### Необхідні передумови

Засвоєння Теми 3 «Циклічні програми»

### Мета завдання

Навчитись писати циклічні програми, що містять ведення з клавіатури, присвоєння та виведення, розгалуження, цикли. Також навчитись писати циклічні програми для графічного зображення простих геометричних фігур з використанням `turtle`.

### Обмеження на виконання завдання

Не використовувати функції.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Розділити обчислення та графічне відображення, потім об'єднати результати.

### Зауваження щодо вирішення завдання

Для розв'язання другої задачі (дано точку, у яку повинен влучити снаряд; розрахувати початкову швидкість та кут) зафіксувати один параметр, наприклад, швидкість, та знайти другий (кут).

Для графічного відображення визначити початкові координати  $x\_start$ ,  $y\_start$  (наприклад, лівого нижнього кута) та масштабний коефіцієнт відображення 1 метру у  $N$  пікселях. Використовувати цей коефіцієнт у всіх діях з `turtle`.

## Групове завдання 4. Знайти задану цифру послідовності.

### Текст завдання

Дано натуральне число  $k$ . Скласти програму одержання  $k$ -тої цифри послідовності

149162536 ... ,

у якій виписані підряд квадрати всіх натуральних чисел (рядки або списки не використовувати)

Записати перші  $k$  цифр послідовності у оберненому порядку (рядки або списки не використовувати).

## Необхідні передумови

Засвоєння Теми 4 «Числові типи даних»

## Мета завдання

Навчитись писати програми, які суттєво використовують цілий тип даних (int) у Python.

## Обмеження на виконання завдання

Не використовувати функції, рядки, списки.

## Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо написати частини програми для пошуку k-ї цифри та для показу цифр у оберненому порядку.

## Зауваження щодо вирішення завдання

Для розв'язання другої задачі (записати перші k цифр послідовності у оберненому порядку) звернути увагу на відображення цифри 0 у оберненому порядку, тобто треба не накопичувати число, а показувати всі цифри по мірі їх отримання.

## Рекомендації щодо перевірки результатів

5 цифра має бути 6

18 цифра має бути 0

## Текст програми з розв'язками

```
k = int(input('k='))
digits_count = 0
power_of_10 = 10
digits_in_square = 1
long_number = 0

i = 0

while digits_count < k:
    i += 1
    square = i ** 2
    if square >= power_of_10:
        power_of_10 *= 10
        digits_in_square += 1
    long_number = long_number * power_of_10 + square
    digits_count += digits_in_square

while digits_count > k:
    long_number //= 10
    digits_count -= 1

the_digit = long_number % 10
print('k-th digit', the_digit)
```

```
print('Digits in reversed order')
while long_number > 0:
    reversed_digit = long_number % 10
    long_number //= 10
    print(reversed_digit)
```

## Групове завдання 5. Максимальний паліндром

### Текст завдання

Натуральне число називається паліндромом у системі числення за основою  $b$ , якщо його позиційний запис у цій системі числення однаково читається зліва направо та справа наліво. Скласти програму для обчислення усіх паліндромів серед натуральних чисел у діапазоні від  $k$  до  $n$  у десятковій системі числення.

Скласти програму для обчислення максимального за значенням паліндрому, який є добутком двох натуральних чисел, що містять по  $m$  десяткових цифр.

Показати сам паліндром та обидва множники.

Виконати для  $m$  від 2 до 6.

### Необхідні передумови

Засвоєння Теми 4 «Числові типи даних»

### Мета завдання

Навчитись писати програми, які суттєво використовують цілий тип даних (int) у Python.

### Обмеження на виконання завдання

Не використовувати функції, рядки, списки.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо виконувати першу та другу задачу.

### Зауваження щодо вирішення завдання

Для розв'язання другої задачі (скласти програму для обчислення максимального за значенням паліндрому) звернути увагу на алгоритм отримання паліндромів. Щоб програма завершилась до кінця пари, їх треба отримувати, починаючи з «кінця», тобто з максимального добутку 2 чисел з  $m$  знаків.

### Рекомендації щодо перевірки результатів

Таблиця з очікуваними результатами другої задачі – нижче

Число знаків $m$	Паліндром	Перший множник	Другий множник
2	9009	99	91
3	906609	993	913
4	99000099	9999	9901

5	9966006699	99979	99681
6	999000000999	999999	999001

## Текст програми з розв'язками

```

power = int(input('m=? '))

lbound = 10 ** (power - 1)
ubound = 10 ** power
pal = 0
k1 = 0
k2 = 0
for i in range(ubound - 1, lbound - 1, -1):
    if i * (ubound - 1) <= pal:
        break

    for j in range(ubound - 1, lbound - 1, -1):
        if i * j <= pal:
            break

        num = i * j
        n = num
        inverse = 0
        while n > 0:
            inverse = inverse * 10 + n % 10
            n //= 10

        if inverse == num:
            pal = num
            k1 = i
            k2 = j

print(pal, k1, k2)

```

## Групове завдання 6. Зображення правильних многокутників

### Текст завдання

Зобразити за допомогою бібліотеки turtle правильні многокутники з різною кількістю сторін та вибрати многокутник з максимальною довжиною сторони.

Многокутники зображувати у випадкових точках екрану з випадково вибраною кількістю сторін.

Одна з сторін многокутника (нижня) паралельна осі абсцис. Для многокутника задається: кількість сторін, довжина сторони (натуральне число), координати точки, з якої починається зображення (лівий нижній кут).

Для роботи також використати бібліотеку random

Для використання turtle треба на початку програми написати

```
import turtle
```

turtle надає графічний курсор для відтворення простих зображень у графічному режимі.

Мінімально потрібні дії з turtle вказано у таблиці

turtle.up()	Підняти пензель догори (припинити малювання)
turtle.down()	Опустити пензель донизу (почати малювання)
turtle.setpos(x, y)	Встановити курсор у позицію (x, y)

Наприклад, щоб зобразити лінію від точки (x1, y1) до точки (x2, y2), треба написати послідовність команд:

```
turtle.up()
```

```
turtle.setpos(x1, y1)
```

```
turtle.down()
```

```
turtle.setpos(x2, y2)
```

Для використання random треба на початку програми написати

```
import random
```

random.randrange(start, stop)	Повертає псевдовипадкове ціле число у діапазоні від start до stop-1
-------------------------------	---

Найбільший многокутник зобразити іншим кольором

Для зміни поточного кольору зображення лінії у random пишуть: turtle.pencolor(cl), де cl – потрібний колір. Колір задається рядком: 'red' або 'blue' або 'green' тощо

Діапазон кількості сторін, довжин сторони та координат точок вибрати самостійно

### Необхідні передумови

Засвоєння Теми 4 «Числові типи даних»

### Мета завдання

Навчитись писати програми, які суттєво використовують дійсний тип даних (float) у Python.

### Обмеження на виконання завдання

Не використовувати функції, списки.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо здійснити генерацію многокутників та зображення одного многокутника.

### Зауваження щодо вирішення завдання

Діапазон кількості сторін многокутників для випадкового вибору не варто задавати більше, ніж 10. Довжину сторони – до 100 пікселів.

Рекомендації щодо розв'язку задачі передбачають використання простих тригонометричних формул для знаходження наступної вершини многокутника. У той же час. turtle має можливості повороту графічного курсору на заданий кут, що може полегшити написання програми. Автор не забороняв користуватись студентам цією можливістю, якщо вони її знали або опанували безпосередньо під час заняття.

### Рекомендації щодо перевірки результатів

Вивести на екран у текстовому режимі кількість сторін та довжину сторони кожного многокутника, а також ці параметри для максимального многокутника.

## Групове завдання 7. Гра у відгадування слова

### Текст завдання

Слова у рядку розділяються одним або декількома пропусками. Скласти програму, яка обчислює кількість слів у рядку та знаходить n-те слово рядка (списки не використовувати).

Використати цю програму для створення гри у відгадування слова.

Гра полягає у наступному:

1. Грають комп'ютер та гравець.
2. Комп'ютер випадковим чином вибирає з довгого рядка слово.
3. Комп'ютер пропонує гравцю відгадати слово за задану максимальну кількість кроків  $m$ .
4. На кожному кроці гри комп'ютер показує слово, де усі невідгадані літери, замінюються зірочками (\*). Окрім цього, комп'ютер показує, скільки кроків залишилось до кінця гри, та кількість набраних гравцем балів.
5. Гравець на кожному кроці гри може вибрати введення літери або слова та ввести з клавіатури літеру або слово.
  - a. Якщо гравець вводить літеру
    - i. Якщо літери немає у слові, комп'ютер переходить до наступного кроку гри
    - ii. Якщо літера є у слові, комп'ютер у позиціях, слова, де знаходиться дана літера, показує замість зірочки цю літеру. Гравцю нараховується по 10 балів за кожне входження до слова відгаданої літери. Якщо усі літери у слові відгадані, гравець виграє.
  - b. Якщо гравець вводить слово
    - i. якщо це слово дорівнює заданому, то гравець виграє, та отримує по 10 балів за кожну невідгадану літеру. Якщо ж у слові залишалось більше 2 невідгаданих літер, бали гравця подвоюються.
    - ii. Якщо це слово не дорівнює заданому, гравець програє
6. Якщо за  $m$  кроків гравець не відгадав слово, - він програє.
7. Якщо гравець виграв, комп'ютер має привітати його та показати кількість набраних балів.

Випадковий вибір номера слова реалізувати за допомогою бібліотеки random

Для використання random треба на початку програми написати

```
import random
```

random.randrange(start, stop)	Повертає псевдовипадкове ціле число у діапазоні від start до stop-1
-------------------------------	---

## Необхідні передумови

Засвоєння Теми 5 «Рядки»

## Мета завдання

Навчитись писати програми, які використовують рядки у Python.

## Обмеження на виконання завдання

Не використовувати функції, списки.

## Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо розв'язувати першу та другу задачу.

## Зауваження щодо вирішення завдання

## Рекомендації щодо перевірки результатів

## Текст програми з розв'язками

## Групове завдання 8. Побудова магічних квадратів

### Текст завдання

Квадратна матриця  $n \times n$  з цілих чисел називається типовим (нормальним) магічним квадратом, якщо вона складається з чисел від 1 до  $n^2$ , усі елементи є різними та суми елементів усіх рядків, стовпчиків, головної та побічної діагоналей є однаковими.

Скласти програму, яка перевіряє, чи є матриця магічним квадратом.

Наприклад, один з квадратів  $3 \times 3$

8	1	6
3	5	7
4	9	2

Побудувати магічні квадрати розміром  $n \times n$  ( $n = 4, 5, 7, 8$ ).



Побудову можна виконувати за алгоритмом (наприклад, <http://www.1728.org/magicsq1.htm>) або випадковим чином. Не дозволяється просто вводити готові магічні квадрати, розміщені у мережі.

Випадковий вибір чисел можна реалізувати за допомогою бібліотеки `random`

Для використання `random` треба на початку програми написати

```
import random
```

<code>random.randrange(start, stop)</code>	Повертає псевдовипадкове ціле число у діапазоні від <code>start</code> до <code>stop-1</code>
<code>random.shuffle(t)</code>	“Перемішує” послідовність <code>t</code>

### Необхідні передумови

Засвоєння Теми 6 «Списки»

### Мета завдання

Навчитись писати програми, які використовують списки у Python.

### Обмеження на виконання завдання

Не використовувати функції.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо розв'язувати першу та другу задачу.

## Групове завдання 9. Перевірка, чи є текст віршом

### Текст завдання

З клавіатури вводять список рядків українською мовою з позначенням наголосів у словах.

Перевірити, чи є цей список рядків віршом.

Будемо вважати, що список є віршом, якщо у ньому витримується ритм та рима.

Для аналізу ритму та рими слід виділити у словах склади. Кожний склад містить рівно одну голосну літеру.

Будемо вважати, що ритм витримується, якщо по складах маємо однакову розстановку наголосів у всіх рядках списку. Наприклад, наголос на 2, 4, 6 склад рядка тощо.

Будемо вважати, що рима витримується, якщо остання голосна літера, на яку падає наголос, у двох сусідніх (або через один) рядках є однаковою та всі приголосні літери після неї до кінця рядка також однакові (якщо після приголосних у кінці рядка йдуть голосні літери, вони можуть відрізнятись).

При введенні списку рядків наголос перед відповідною голосною літерою позначати символом «`» - обернений апостроф.

Наприклад, при введенні список рядків віршу може виглядати так:

Вже п`очал`ось, маб`уть, майб`утнє.

Оц`е, либ`онь, вже п`очал`ось...

Не з`абув`айте н`езаб`утнє,

Вон`о вже `іне`єм взял`ось!

### Необхідні передумови

Засвоєння Теми 6 «Списки».

### Мета завдання

Навчитись писати програми, які використовують списки та взаємозв'язок між рядками та списками у Python.

### Обмеження на виконання завдання

Не використовувати функції.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо розв'язувати задачі перевірки ритму та рими.

### Зауваження щодо вирішення завдання

### Рекомендації щодо перевірки результатів

Перевірити різні варіанти наявності або відсутності рими та ритму. Зокрема, для перевірки рими у даному прикладі третій рядок можна подати у вигляді:

«Не з`абув`айте н`езаб`утно» - є рима

або

«Не з`абув`айте н`езаб`утко» - немає рими

## Групове завдання 10. Аналіз шахової позиції

### Текст завдання



Шахова позиція задається словником, у якому для непорожніх клітинок шахової дошки вказують фігури, які на них розташовані. Ключами є кортежі (вертикаль, горизонталь), а значеннями – кортежі з назвою та кольором фігури. Наприклад, ('h', 7): ('Кінь', 'білий')

Ввести з клавіатури шахову позицію та зобразити її за допомогою бібліотеки turtle.

Проаналізувати позицію, та з'ясувати, чи є у ній шах чорному королю.

Для зображення позиції використати символи юнікод для шахових фігур:

Вид	Числовий код	Опис
♚	9818;	Чорний король
♛	9819;	Чорний ферзь
♜	9820;	Чорна тура

	9821;	Чорний слон
	9822;	Чорний кінь
	9823;	Чорний пішак
	9812;	Білий король
	9813;	Білий ферзь
	9814;	Біла тура
	9815;	Білий слон
	9816;	Білий кінь
	9817;	Білий пішак

Для зображення рядка “abc” у turtle можна використати команду, де 12 – розмір шрифту:

```
turtle.write("abc", font=("", 12))
```

Інші дії з turtle вказано у таблиці

turtle.up()	Підняти пензель догори (припинити малювання)
turtle.down()	Опустити пензель донизу (почати малювання)
turtle.setpos(x, y)	Встановити курсор у позицію (x, y)

Для зображення одного зафарбованого квадрату дошки довжиною *a* можна використати такі команди turtle:

```
turtle.color('black', 'light grey')
```

```
turtle.begin_fill()
```

```
for i in range(4):
```

```
    turtle.fd(a)
```

```
    turtle.right(90)
```

```
turtle.end_fill()
```

### Короткі правила гри у шахи



Кожна з 6 фігур ходить по-різному. Фігури не можуть перестрибувати через інші фігури (робити це може тільки кінь), ніколи не можуть вставати на клітку, де вже стоїть фігура того ж (свого) кольору. Однак вони можуть вставати на місце фігури супротивника, яку вони захоплюють (беруть в полон).

Король - найважливіша фігура, але при цьому і одна з найслабших. Король може ходити тільки на одну клітку в будь-якому напрямку - вгору, вниз, в сторони і по діагоналі. Король ніколи не може ходити на клітку, яка знаходиться під шахом (де його може взяти фігура суперника). Коли король атакований інший фігурою, це називається "шах".

Тура може ходити на будь-яке число клітин, але тільки вперед, назад і в сторони (не по діагоналі).

Слон може ходити по прямій на будь-яке число клітин, але лише по діагоналі. Протягом гри кожен слон завжди ходить по клітинам одного і того ж кольору (світлим або темним).

Коні ходять інакше, ніж інші фігури - на дві клітини в одному напрямку і далі на одну клітку під кутом 90 градусів. Хід коня нагадує букву "Г". Кінь - єдина фігура, яка, роблячи хід, може перестрибувати через інші фігури.

Пішак - незвичайна фігура, вона ходить і бере по-різному: ходити пішак може лише вперед, а брати - лише по діагоналі. Пішак може пересуватися тільки на одну клітку за один хід, крім самого першого ходу, коли він може сходити вперед на одну або на дві клітини. Пішак може брати тільки по діагоналі на одну клітку перед собою. Пішак не може ходити або брати назад.

У пішака є одна чудова особливість - якщо він доходить до протилежного боку дошки, він може стати будь-який інший фігурою (це називається "перетворенням пішака"). Пішак може перетворитися в будь-яку фігуру.

### Необхідні передумови

Засвоєння Теми 7 «Кортежі» та Теми 8 «Словники».

### Мета завдання

Навчитись писати програми, які використовують кортежі та словники у Python.

## Обмеження на виконання завдання

Не використовувати функції.

## Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо розв'язувати задачі відображення та аналізу шахової позиції.

## Зауваження щодо вирішення завдання

Завдання є доволі складним та об'ємним. Тому його слід давати тільки якщо група є добре підготовленою. Інакше дане завдання можна опустити.

## Рекомендації щодо перевірки результатів

Задати прості позиції, в яких є або немає шаху чорному королю.

## Групове завдання 11. Перевірка, чи є рядок правильним виразом

### Текст завдання

Скласти програму, яка перевіряє, чи є заданий рядок правильним записом виразу у Python, який містить

імена змінних,

знаки операцій: +, -, \*, /, %, (мінус бінарний, тобто тільки  $a - b$ , а не  $-a$  або  $-b$ ),

круглі дужки.

Рядок також може містити пропуски.

Вважати, що імена змінних складаються з 1 символу.

Перевірити, що:

- Рядок містить тільки допустимі для виразу символи: великі або маленькі латинські літери, знаки операцій, дужки, пропуски
- Рядок містить тільки допустимі пари символів (наприклад, «змінна» - «операція»)
- У рядку правильно розставлені дужки
- Рядок починається та закінчується правильним символом (літерою або дужкою)

Побудувати функції для кожного пункту перевірки та показати результат цих перевірок для заданого виразу, а також загальний результат: чи є рядок правильним записом виразу.

Одним з можливих розв'язків є типізація усіх можливих символів, тобто, поділ їх на типи: «змінна», «операція», «відкриваюча дужка», «закриваюча дужка».

Для розв'язку також можуть знадобитись додаткові структури даних: рядки, списки, словники, кортежі.

## Необхідні передумови

Засвоєння Теми 9 «Підпрограми» а також попередніх тем курсу.

## Мета завдання

Навчитись писати програми, які використовують функції та структури даних у Python.

## Обмеження на виконання завдання

Не використовувати класи.

## Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо писати функції перевірки, оскільки вони є незалежними.

## Зауваження щодо вирішення завдання

В якості згаданих структур даних може бути словник типів символів ('+' – операція тощо). Також корисним буде словник допустимих пар символів та рядок допустимих символів (літери, цифри знаки операцій, дужки).

## Рекомендації щодо перевірки результатів

Перевірити варіанти неправильного виразу для кожної функції а також правильних виразів.

## Групове завдання 12. Пошук виходу з лабіринту

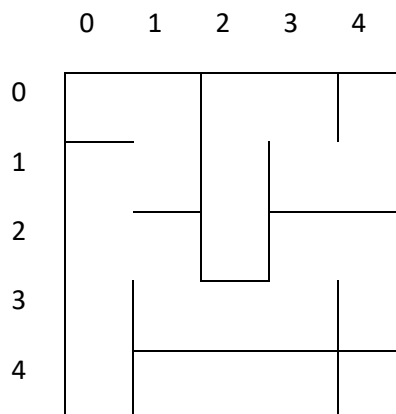
### Текст завдання

Квадратний лабіринт є сукупністю кімнат, між якими можуть бути проходи. Є також один вихід з лабіринту.

Лабіринт задано матрицею розміром  $n \times m$ , елементи якої  $a_{ij}$  – кортежі з 4 значень 0 або 1. Якщо на певному місці стоїть 0, це означає, що стіни немає, 1 – стіна є. Перший елемент кортежу – верх, другий – праворуч, третій – низ, четвертий – ліворуч.

Наприклад, кортеж (1, 0, 1, 1) означає, що у даній кімнаті є усі стіни, окрім стіни праворуч.

Приклад лабіринту показано нижче:



Нехай задано лабіринт та початкове положення (кімната, що задається індексами рядка та стовпчика) у ньому.

Зобразити цей лабіринт за допомогою бібліотеки turtle.

Скласти програму, яка знаходить вихід з лабіринту або повідомляє, що вийти з даної кімнати неможливо. Показати шлях виходу з лабіринту на зображенні лабіринту або у вигляді послідовності пройдених кімнат.

Щоб використати turtle, слід на початку програми написати

```
import turtle
```

Дії над turtle:

turtle.up()	Підняти пензель догори (припинити малювання)
turtle.down()	Опустити пензель донизу (почати малювання)
turtle.setpos(x, y)	Встановити курсор у позицію (x, y)
turtle.fd(c)	Перейти уперед на c точок
turtle.right(alpha)	Повернутись праворуч на кут alpha (у градусах)
turtle.left(alpha)	Повернутись ліворуч на кут alpha (у градусах)

Для зображення однієї кімнати лабіринту, якщо відповідний кортеж зі стінами – це x - та графічний курсор знаходиться у лівому верхньому куті квадрату з орієнтацією праворуч, можна використати такі команди turtle:

```
for w in x:
```

```
    if w == 1:
```

```
        turtle.down()
```

```
    turtle.fd(c)
```

```
    turtle.right(90)
```

```
    turtle.up()
```

### Необхідні передумови

Засвоєння Теми 9 «Підпрограми» а також попередніх тем курсу.

### Мета завдання

Навчитись писати програми, які використовують функції (зокрема, рекурсивні) та структури даних у Python.

### Обмеження на виконання завдання

Не використовувати класи.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо писати функції для знаходження шляху та для зображення лабіринту.

### Зауваження щодо вирішення завдання

Для знаходження шляху можна використовувати рекурсивну функцію по аналогії з пошуком туру коня (один з прикладів у матеріалах лекцій до Теми 9).

### Рекомендації щодо перевірки результатів

Перевірити варіанти з існуючим та неіснуючим шляхом.

## Групове завдання 13. Дешифрування повідомлення

### Текст завдання

Слідчі перехопили шифроване повідомлення ворожого агента.

Повідомлення складалось із чисел:

1315 16 423 20 39 1 1359 28 309 2 441 31

Коли слідчі зайшли у помешкання, де жив агент, вони знайшли на полиці три книги та припустили, що у повідомленні парами чисел зашифровані слова з цих книг таким чином, що перше число- це номер абзацу, а друге – номер слова у абзаці.

Треба дешифрувати повідомлення.

Для пришвидшення дешифрування скористатись електронним варіантом книжок. Абзац – це текст, який завершується переведенням рядка ('\n'), за яким йде принаймні один порожній рядок, що також закінчується переведенням рядка. Порожніх рядків у кінці абзацу може бути й більше, ніж один.

Слова можуть розділятися пропусками та розділовими знаками: '.', ',', ':', ';', '-', '!', '?', '"', "'", тире '—' (код символу 8212), лапки '«' (код 171) та '»' (код 187), , багато крапок '...' (код 8230). Розділові знаки не є словами. Нумерація абзаців та слів у абзацах починається з 1.

Книги збережені у кодуванні 'utf-8'. Тобто відкривати їх треба так:

```
f = open(filename, 'r', encoding='utf-8')
```

Скласти програму для дешифрування та показати дешифроване повідомлення.

### Необхідні передумови

Засвоєння Теми 12 «Файли».

### Мета завдання

Навчитись писати програми, які використовують текстові файли у Python.

### Обмеження на виконання завдання

Не використовувати класи.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо писати функції для знаходження абзаців та слів а також будувати змістовні повідомлення зі слів.



### Зауваження щодо вирішення завдання

Номери абзаців та слів у тексті вище є умовними. Для завдання треба підібрати у мережі 3 тексти, які вільно розповсюджуються (файли у форматі .txt). Треба перевірити, що файли мають вигляд, наведений у тексті завдання (після кожного абзацу – порожній рядок. Бажано, щоб тексти були книгами з детективним сюжетом.

Далі змінити імена файлів на 1.txt, 2.txt, 3.txt. Придумати текст майбутнього повідомлення та послідовно за допомогою запуску програми у режимі пошуку слів (нижче) знайти номери абзаців та слів з повідомлення. Бажано задати якесь правило регулярного підбору слів з текстів. Наприклад, перше слово – з 1.txt, друге – з 2.txt тощо.

Після того, як номери абзаців та слів виписані, перевірити правильність повідомлення за допомогою запуску програми у режимі дешифрування (нижче).

Перед заняттям розіслати студентам три текстові файли.

Бажано щорічно змінювати текст повідомлення та/або джерела.

### Рекомендації щодо перевірки результатів

Команда, яка зробила завдання, має просто написати або показати повідомлення. Перевірити, що результат отримано за допомогою програми, а не підбором.

### Текст програми з розв'язками

Програма пошуку слів та дешифрування. Завдання студентам включає тільки частину цієї програми, яка здійснює дешифрування. Для роботи програми у режимі пошуку після запуску ввести 1. Для роботи програми у режимі дешифрування після запуску ввести 2.

У режимі пошуку також ввести слово. Програма повертає список кортежів <№ абзацу, № слова> усіх входжень слова у всі тексти.

У режимі дешифрування також ввести номер абзацу та номер слова. Програма має повернути дане слово в усіх 3 текстах. Якщо такого абзацу та/або номеру слова немає у деякому тексті, - повертає «no para»/ «no word».

```
#!/usr/bin/env python3
import re

EMPTY_LINE = '\n'

def get_next_paragraph(lines, i):
    while i < len(lines) and lines[i] == EMPTY_LINE:
        i+=1

    para = []
    while i < len(lines) and lines[i] != EMPTY_LINE:
        para.append(lines[i])
        i+=1

    return para, i
```

```

def get_word(para, word, para_no):
    words = []
    for line in para:
        string = re.sub(r'["'?!?.,+;:"'() \-...-«»]+', ' ', line)
        string = string.lower()
        words += string.split()
#         print(words)

    word_indeces = []
    while word in words:
        i = words.index(word) + 1
        words = words[i:]
        word_indeces.append((para_no, i))

    return word_indeces

def get_all_words(lines, word):
    para_no = 0
    line_no = 0
    indeces = []
    while True:
        para, line_no = get_next_paragraph(lines, line_no)
#         print(para)
        if not para:
            break
        para_no += 1
        indeces += get_word(para, word, para_no)

    return indeces

def show_word_occures():
    word = input("word: ")
    for filename in files:
        with open(filename, 'r', encoding='utf-8') as f:
            lines = f.readlines()
            ind = get_all_words(lines, word)
            print(filename)
            for j, i in enumerate(ind):
                if j > 10:
                    break
            print(i)

def test_para_word_no():
    para_no = int(input("paragraph no: "))

```

```

word_no = int(input("word no: "))

for filename in files:
    with open(filename, 'r', encoding='utf-8') as f:
        lines = f.readlines()

        line_no = 0
        for i in range(para_no):
            para, line_no = get_next_paragraph(lines, line_no)
            if line_no >= len(lines):
                print("no para")
                break

        words = []
        for line in para:
            string = re.sub(r' '[!?.,+;'"()\-...-«»]+''', ' ',
line)

            string = string.lower()
            words += string.split()

        print(filename)

        print(words[word_no - 1] if word_no <= len(words) else
"no word")

# hardcode paths to 3 texts
files = ["1.txt",
        "2.txt",
        "3.txt"]

mode = int(input("mode [1 - word, 2 - test]"))

if mode == 1:
    show_word_occures()
else:
    test_para_word_no()

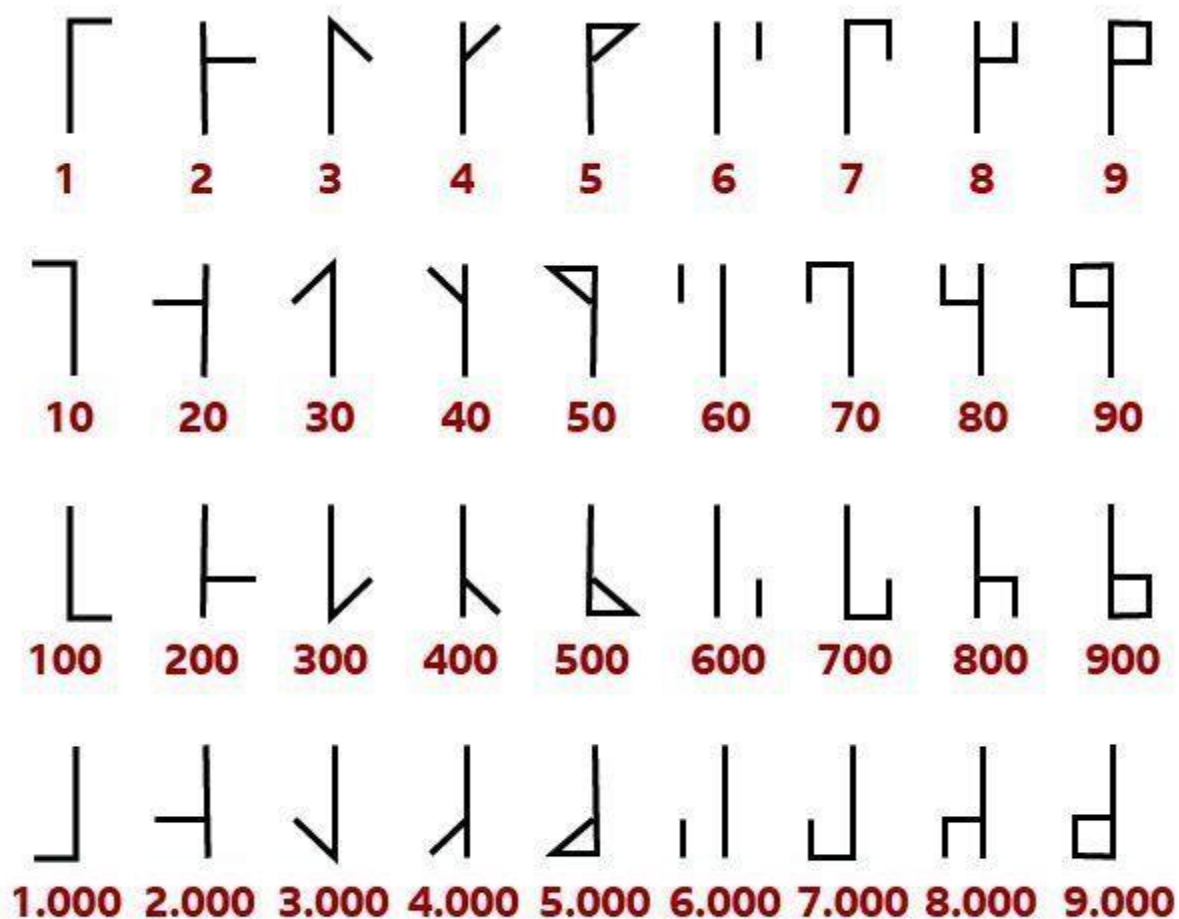
```

## Групове завдання 14. Цистеріанські числа

### Текст завдання

Цистеріанська система числення дозволяє зображувати числа до 4 десяткових знаків одним цистеріанським знаком.

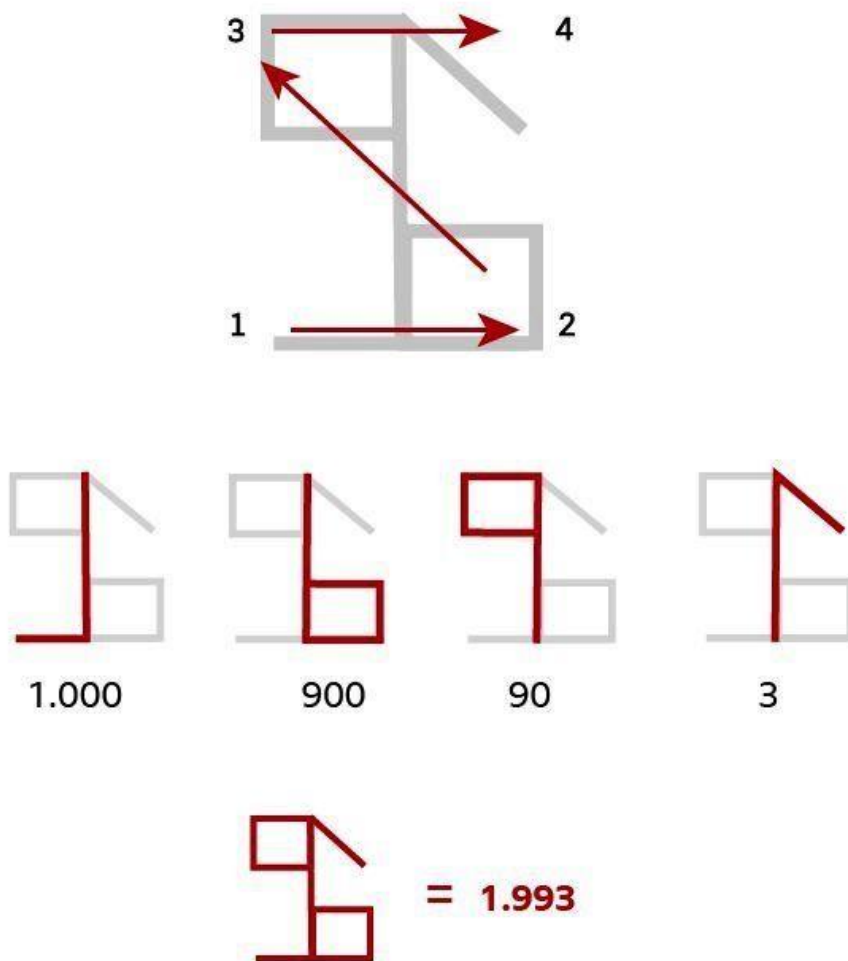
Цистеріанські знаки для чисел, що визначають одиниці, десятки, сотні та тисячі, виглядають так:



BBC

Для отримання довільного чотиризначного числа їх комбінують.

На малюнку (див. нижче) кожен сектор або квадрант містить зображення тисяч (1), сотень (2), десятків (3) та одиниць (4) у наступному порядку:



BBC

**Завдання:** описати клас CisterianNumber для числа у цистеріанській системі числення. У цьому класі передбачити поля

\_number – число

\_digits – список десяткових цифр числа

Окрім конструктора, який створює цистеріанське число за заданим числом n, передбачити також методи:

- додавання до числа іншого цистеріанського числа,
- різниці числа з іншим цистеріанським числом,
- зображення цистеріанського числа за допомогою turtle.

Для зображення задати масштаб по горизонталі та вертикалі. Якщо при виконанні арифметичних операцій результат виходить за межі 4 знаків або є від'ємним, - повертати його у діапазон від 0 до 9999

З використанням класу CisterianNumber розв'язати задачу: вводиться послідовність натуральних чисел до 4 знаків. Утворити з кожного числа цієї послідовності цистеріанське число, показати ці числа у «рядок» у вікні turtle, обчислити їх суму та показати її у наступному «рядку».

Відступи для чисел у одному «рядку» та між «рядками» можна робити у половину розміру числа.

Підказка: відповідні десяткові цифри для одиниць, десятків, сотень та тисяч очевидно є симетричними по горизонталі та вертикалі. Це можна використати для зображення чисел.

Більш докладно про цистеріанську систему числення можна прочитати у <https://www.bbc.com/ukrainian/features-55969229>

## Необхідні передумови

Засвоєння Теми 13 «Класи та об'єкти».

## Мета завдання

Навчитись писати програми, які використовують класи у Python.

## Обмеження на виконання завдання

Немає.

## Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо писати код класу для цистеріанського числа та їх суми та для зображення числа у turtle.

## Зауваження щодо вирішення завдання

Для розв'язку стануть у пригоді додаткові символічні константи та типи даних. Найбільша складність у цьому завданні – компактна реалізація зображення цистеріанського числа.

Представимо кожне число як список з 4 десяткових цифр. Кожна цифра на зображенні представлена як набір елементів: верхня горизонталь, нижня горизонталь, діагональ, що опускається, діагональ, що піднімається, вертикаль. Кожна цифра може бути представлена кортежем з цих елементів (словник ELEMENTS у програмі нижче). Порядок цифри вказує на квадрант, у якому треба зобразити цифру та на необхідність симетричного відображення по вертикалі та/або горизонталі.

## Рекомендації щодо перевірки результатів

Запропонувати приклади з сумою чисел з 1, 2, 3 та 4 знаків.

## Текст програми з розв'язками

Опис класу . CisterianNumber (модуль cisterian).

```
import turtle
```

```
SCALE_X = 40  
SCALE_Y = 60
```

```
HOR_UP = 1  
HOR_DOWN = 2
```

```

DESCEND = 3
ASCEND = 4
VERT = 6
ELEMENTS = {
    0: (),
    1: (HOR_UP, ),
    2: (HOR_DOWN, ),
    3: (DESCEND, ),
    4: (ASCEND, ),
    5: (HOR_UP, ASCEND),
    6: (VERT, ),
    7: (HOR_UP, VERT),
    8: (HOR_DOWN, VERT),
    9: (HOR_UP, HOR_DOWN, VERT)
}

class CisterianNumber:
    def __init__(self, number):
        self._number = number
        self._digits = []
        n = self._number
        for i in range(4):
            self._digits.append(n % 10)
            n //= 10

    def add(self, other):
        return CisterianNumber((self._number + other._number) %
10000)

    def sub(self, other):
        return CisterianNumber(max(self._number - other._number,
0))

    def show(self, x, y):
        self._show_base(x, y)
        for order in range(4):
            self._show_digit(self._digits[order], order + 1, x,
y)

    def _line(self, x1, y1, x2, y2):
        turtle.up()
        turtle.setpos(x1, y1)
        turtle.down()
        turtle.setpos(x2, y2)

    def _show_base(self, x, y):
        x1 = x2 = x + SCALE_X // 2

```

```

y1 = y
y2 = y - SCALE_Y
self._line(x1, y1, x2, y2)

def _mirror_hor(self, x, xx):
    delta = xx - (x + SCALE_X // 2)
    return (x + SCALE_X // 2) - delta

def _mirror_vert(self, y, yy):
    delta = yy - (y - SCALE_Y // 2)
    return (y - SCALE_Y // 2) - delta

def _define_coords(self, element, order, x, y):
    if element == HOR_UP:
        x1 = x + SCALE_X // 2
        x2 = x + SCALE_X
        y1 = y2 = y
    elif element == HOR_DOWN:
        x1 = x + SCALE_X // 2
        x2 = x + SCALE_X
        y1 = y2 = y - SCALE_Y // 3
    elif element == DESCEND:
        x1 = x + SCALE_X // 2
        x2 = x + SCALE_X
        y1 = y
        y2 = y - SCALE_Y // 3
    elif element == ASCEND:
        x1 = x + SCALE_X // 2
        x2 = x + SCALE_X
        y1 = y - SCALE_Y // 3
        y2 = y
    elif element == VERT:
        x1 = x2 = x + SCALE_X
        y1 = y
        y2 = y - SCALE_Y // 3

    if order in {2, 4}:
        x1 = self._mirror_hor(x, x1)
        x2 = self._mirror_hor(x, x2)
    if order in {3, 4}:
        y1 = self._mirror_vert(y, y1)
        y2 = self._mirror_vert(y, y2)
    return x1, y1, x2, y2

def _show_digit(self, digit, order, x, y):
    for element in ELEMENTS[digit]:
        x1, y1, x2, y2 = self._define_coords(element, order,

```



```

x, y)
        self._line(x1, y1, x2, y2)

if __name__ == '__main__':
    n = int(input('n=? '))
    c = CisterianNumber(n)
    c.show(-100, 100)
    m = input()

```

Головний модуль

```

from cisterian import CisterianNumber, SCALE_X, SCALE_Y

numbers = list(map(int, input('numbers? ').split()))
s = CisterianNumber(0)
start_x = x = -200
y = 200
for i in numbers:
    c = CisterianNumber(i)
    c.show(x, y)
    s = s.add(c)
    x += SCALE_X + SCALE_X // 2
y -= (SCALE_Y + SCALE_Y // 2)
s.show(start_x, y)
k = input()

```

## Групове завдання 15. Інтерпретатор лінійних програм. Крок 1

### Текст завдання

Треба розробити модулі `tokenizer`, `syntax_analyzer`, `storage` а також наведені класи згідно специфікації.

Специфікація для пришвидшення розробки надається у вигляді «кістяка» модулів у файлах `.py` у окремому архіві.

Після розробки кожного модуля його треба запустити окремо та досягти виведення значення

Success = True

Основну частину кожного модуля після

```
if __name__ == "__main__":
```

не змінювати (можна вставляти `print` для налагодження)

Усі функції модуля мають бути реалізовані

Заголовки функцій не змінювати

Можна додавати власні внутрішні функції у модулі, якщо потрібно.

Після завершення розробки модулів запустити модуль main та впевнитись, що виводиться

Success = True

Для неповних команд (з 2 або одного студента) достатньо реалізувати частину модулів по порядку (tokenizer, syntax\_analyzer)

### Необхідні передумови

Засвоєння Теми 13 «Класи та об'єкти».

### Мета завдання

Навчитись писати програми за заданою специфікацією у Python.

### Обмеження на виконання завдання

Немає.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо писати код модулів tokenizer, syntax\_analyzer, storage. Оскільки модуль tokenizer використовується у syntax\_analyzer, рекомендувати командам задіяти кращих студентів у його написанні.

### Зауваження щодо вирішення завдання

Це завдання є першим з 3 послідовних завдань, мета яких – написати власний інтерпретатор, який дозволить виконувати прості лінійні програми у мові, подібній до Python. Особливістю цих 3 завдань є пророблена жорстка специфікація модулів та класів а також наявність розроблених тестів, які дозволяють миттєво перевірити правильність розроблених студентами програм. Команди мають «вбудувати» розроблені ними фрагменти програм у частково готові модулі та класи. Окрім опанування такого стилю роботи (розробка програм за специфікаціями) студенти також бачать фрагмент розробленого викладачем програмного коду, який дає уявлення про те, що саме очікується від розробки.

Перед заняттям студентам розсилається «кістяк» модулів, які треба наповнити програмним кодом. У кожному модулі є документація щодо розроблюваної функціональності. Кістяк модулів для даного заняття наведений після повного тексту програми.

Також студенти повинні мати можливість після заняття і до наступного заняття обмінюватись працюючими модулями, які знадобляться для наступного заняття.

### Рекомендації щодо перевірки результатів

Перевірити правильність виконання тестів окремих модулів та головного модуля.

### Текст програми з розв'язками

Модуль . tokenizer

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

"""

*Модуль призначено для синтаксичного розбору виразу та присвоєння*

по частинах.

Вираз може мати вигляд:

`(abc + 123.5)*d2-3/(x+y)`

присвоєння може мати вигляд:

`x = a + b`

Вираз може містити:

змінні - ідентифікатори

константи - дійсні або цілі числа без знаку

знаки операцій: +, -, \*, /

дужки: (, )

Функція `get_tokens` за заданим виразом має повертати

послідовність лексем - tokenів

Кожний token - це кортеж: (<тип токена>, <значення токена>)

"""

```
from collections import namedtuple
```

```
# типи tokenів
```

```
TOKEN_TYPE = ("variable",  
              "constant",  
              "operation",  
              "left_paren",  
              "right_paren",  
              "other",  
              "equal")
```

```
# словник фіксованих tokenів, що складаються з одного символу
```

```
TOKEN_TYPES = {"+": "operation",  
               "-": "operation",  
               "*": "operation",  
               "/": "operation",  
               "(": "left_paren",  
               ")": "right_paren",  
               "=": "equal"}
```

```
# тип токена
```

```
Token = namedtuple('Token', ['type', 'value'])
```

```
def get_tokens(string):
```

```
    """
```

```
    Функція за рядком повертає список tokenів типу Token
```

```
    :param string: рядок
```

```
    :return: список tokenів
```

```
    """
```

```

tokens = []
while string:
    next_tok, string = _get_next_token(string)
#    print(next_tok, string)
    if next_tok:
        tokens.append(next_tok)
return tokens

def _get_next_token(string):
    """
    Функція повертає наступний токен та залишок рядка
    Використовує внутрішні функції _get_constant, _get_variable
    :param string: рядок
    :return: next_tok - наступний токен, якщо є, або None
    :return: string - залишок рядка
    """
    string = string.strip()
    if not string:
        return None, string

    if string[0] in TOKEN_TYPES:
        next_tok = Token(TOKEN_TYPES[string[0]], string[0])
        string = string[1:]
    else:
        num, s = _get_constant(string)
        if num:
            next_tok = Token("constant", num)
            string = s
        else:
            var, s = _get_variable(string)
            if var:
                next_tok = Token("variable", var)
                string = s
            else:
                next_tok = Token("other", string[0])
                string = string[1:]

    return next_tok, string

def _get_constant(string):
    """
    Функція за рядком повертає константу (якщо є) та залишок
    рядка
    :param string: рядок
    :return: константа (або порожній рядок), залишок рядка
    """

```

```

    if not string or not string[0].isdigit():
        return "", string

    k = 0
    for i, c in enumerate(string):
        if not c.isdigit() and c != '.' or c == '.' and k > 1:
            break
        if c == '.':
            k += 1
    else:
        i += 1

    return string[:i], string[i:]

def _get_variable(string):
    """
    Функція за рядком повертає змінну (якщо є) та залишок рядка
    :param string: рядок
    :return: змінна (або порожній рядок), залишок рядка
    """
    if not string or not string[0].isalpha() and string[0] != '_':
        return "", string

    for i, c in enumerate(string):
        if not c.isalnum() and c != '_':
            break
    else:
        i += 1

    return string[:i], string[i:]

if __name__ == "__main__":
    success = get_tokens("((ab1_ - 345.56) (*.2{_cde23") == (
        Token(type='left_paren', value='('),
        Token(type='left_paren', value='('),
        Token(type='left_paren', value='('),
        Token(type='variable', value='ab1_'),
        Token(type='operation', value='-'),
        Token(type='constant', value='345.56'),
        Token(type='right_paren', value=')'),
        Token(type='left_paren', value='('),
        Token(type='operation', value='*'),
        Token(type='operation', value='/'),
        Token(type='other', value='.'),
    )

```

```

Token(type='constant', value='2'),
Token(type='other', value='{'),
Token(type='variable', value='_cde23')]

success = success and get_tokens("x = (a + b)")==[
    Token(type='variable', value='x'),
    Token(type='equal', value('='),
    Token(type='left_paren', value='('),
    Token(type='variable', value='a'),
    Token(type='operation', value='+'),
    Token(type='variable', value='b'),
    Token(type='right_paren', value=')')]
print("Success =", success)

```

### Модуль syntax\_analyzer

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""

```

Модуль призначено для перевірки синтаксичної правильності виразу та присвоєння.

Вираз може мати вигляд:

$(abc + 123.5) * d2 - 3 / (x + y)$

Вираз може містити:

змінні - ідентифікатори

константи - дійсні або цілі числа без знаку

знаки операцій: +, -, \*, /

дужки: (, )

Присвоєння - це

$\langle \text{змінна} \rangle = \langle \text{вираз} \rangle$

наприклад

$x = a + b$

Метод `check_expression_syntax` за заданим списком токенів для виразу має повернути

булівське значення та (можливо) помилку

Кожний токен - це кортеж:  $\langle \text{тип токена} \rangle, \langle \text{значення токена} \rangle$

Перевірка робиться на допустимість сусідніх токенів, правильний перший та останній токен, порожній вираз, правильність розставлення дужок

Метод `check_assignment_syntax` за заданим списком токенів для присвоєння має повернути

булівське значення та (можливо) помилку.

```

"""
from tokenizer import Token, get_tokens

# СЛОВНИК МНОЖИН ДОПУСТИМИХ НАСТУПНИХ ТОКЕНІВ ДЛЯ ЗАДАНОГО
# ТОКЕНА
VALID_PAIRS = {"variable": {"operation", "right_paren"},
               "constant": {"operation", "right_paren"},
               "operation": {"variable", "constant",
                             "left_paren"},
               "left_paren": {"left_paren", "variable",
                              "constant"},
               "right_paren": {"right_paren", "operation"},
               "other": set()}

# СЛОВНИК ПОМИЛОК
ERRORS = {"invalid_pair": "Недопустима пара токенів {}, {}",
          "incorrect_parens": "Неправильно розставлені дужки",
          "empty_expr": "Порожній вираз"}

class SyntaxAnalyzer:
    def __init__(self, tokens):
        self._tokens = tokens[:]

    def check_expression_syntax(self):
        """
        Метод перевіряє синтаксичну правильність виразу за
        списком токенів.
        Використовує внутрішні методи _check_parens, _check_pair
        Повертає булівське значення та рядок помилки.
        Якщо помилки немає, то повертає порожній рядок
        :param tokens: список токенів
        :return: success - булівське значення
        :return: error - рядок помилки
        """
        self._tokens = [Token("left_paren", "(")] + self._tokens
        + [Token("right_paren", ")")]
        if len(self._tokens) == 2:
            return False, ERRORS["empty_expr"]

        if not self._check_parens():
            return False, ERRORS["incorrect_parens"]

        for i in range(len(self._tokens) - 1):
            if not self._check_pair(self._tokens[i],
self._tokens[i + 1]):
                return False,
ERRORS["invalid_pair"].format(self._tokens[i],

```

```

self._tokens[i + 1])
    return True, ""

def _check_parens(self):
    """
    Метод перевіряє чи правильно розставлені дужки у виразі.
    Повертає булівське значення
    :param tokens: список токенів
    :return: success - булівське значення
    """
    depth = 0
    for tok in self._tokens:
        if tok.type == "left_paren":
            depth += 1
        elif tok.type == "right_paren":
            depth -= 1
        if depth < 0:
            break

    return depth == 0

def _check_pair(self, tok, next_tok):
    """
    Метод перевіряє чи правильна пара токенів.
    Повертає булівське значення
    :param tok: поточний токен
    :param next_tok: наступний токен
    :return: success - булівське значення
    """

    return next_tok.type in VALID_PAIRS[tok.type]

if __name__ == "__main__":
    analyzer = SyntaxAnalyzer(get_tokens("((ab1_ -
345.56) (*.2{ _cde23"))
    success1, error1 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzer(get_tokens("(ab1_ -
345.56) (*.2 _cde23"))
    success2, error2 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzer(get_tokens("- 345.56 (*.2 _cde23"))
    success3, error3 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzer(get_tokens("2 - 345.56 *"))
    success4, error4 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzer(get_tokens("2 - .2"))

```



```

    success5, error5 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzer(get_tokens(" "))
    success6, error6 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzer(get_tokens("((abc -3 * b2) + d5 /
7)"))
    success7, error7 = analyzer.check_expression_syntax()

    success = (
        not success1 and error1 == 'Неправильно розставлені
дужки' and
        not success2 and error2 ==
        "Недопустима пара токенів Token(type='operation',
value='*'), "
        " Token(type='operation', value='/') " and
        not success3 and not success4 and
        not success5 and error5 ==
        "Недопустима пара токенів Token(type='operation',
value='-'), "
        "Token(type='other', value='.') " and
        not success6 and error6 == "Порожній вираз" and
        success7 and error7 == ""
    )

    print("Success =", success)

```

## Модуль storage

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Модуль призначено для реалізації пам'яті, що складається зі
змінних.

Змінні можуть мати числові значення цілого або дійсного типу

"""

# словник, що співствляє коди помилок до їх описи
ERRORS = {0: "",
          1: "Змінна вже є у пам'яті",
          2: "Змінна не існує",
          3: "Змінна невизначена"}

class Storage():
    def __init__(self):
        self._storage = {}          # пам'ять

```

```

        self._last_error = 0          # КОД ПОМИЛКИ ОСТАННЬОЇ
операції

def add(self, variable):
    """
    Метод додає змінну у пам'ять.
    Якщо така змінна вже існує, то встановлює помилку
    :param variable: змінна
    :return: None
    """
    if variable in self._storage:
        self._last_error = 1
    else:
        self._last_error = 0
        self._storage[variable] = None

def is_in(self, variable):
    """
    Метод перевіряє, чи є змінна у пам'яті.
    :param variable: змінна
    :return: булівське значення (True, якщо є)
    """
    self._last_error = 0
    return variable in self._storage

def get(self, variable):
    """
    Метод повертає значення змінної.
    Якщо така змінна не існує або невизначена (==None),
    то встановлює відповідну помилку
    :param variable: змінна
    :return: значення змінної
    """
    if variable not in self._storage:
        self._last_error = 2
        value = None
    elif self._storage[variable] is None:
        self._last_error = 3
        value = None
    else:
        self._last_error = 0
        value = self._storage[variable]
    return value

def set(self, variable, value):
    """
    Метод встановлює значення змінної

```

```

        Якщо змінна не існує, повертає помилку
        :param variable: змінна
        :param value: нове значення
        :return: None
        """
        if variable not in self._storage:
            self._last_error = 2
        else:
            self._last_error = 0
            self._storage[variable] = value

    def input_var(self, variable):
        """
        Метод здійснює введення з клавіатури та встановлення
        значення змінної
        Якщо змінна не існує, повертає помилку
        :param variable: змінна
        :return: None
        """
        if variable not in self._storage:
            self._last_error = 2
        else:
            self._last_error = 0
            self._storage[variable] = float(input("{} = ?".format(variable)))

    def input_all(self):
        """
        Метод здійснює введення з клавіатури та встановлення
        значення
        усіх змінних з пам'яті
        :return: None
        """
        self._last_error = 0
        for variable in self._storage:
            self.input_var(variable)

    def clear(self):
        """
        Метод видаляє усі змінні з пам'яті
        :return: None
        """
        self._last_error = 0
        self._storage.clear()

    def get_last_error(self):
        """

```

```

        Метод повертає код останньої помилки code
        Для виведення повідомлення треба взяти
        storage.ERRORS[code]
        усіх змінних з пам'яті
        :return: код останньої помилки
        """
        code = self._last_error
        return code

def get_all(self):
    """
    Метод повертає словник змінних пам'яті
    :return: словник змінних
    """
    return self._storage

if __name__ == "__main__":
    store = Storage()
    store.add("a")
    success = store.get_last_error() == 0
    store.add("a")
    success = success and store.get_last_error() == 1
    c = store.get("a")
    success = success and c == None and store.get_last_error()
== 3
    c = store.get("b")
    success = success and c == None and store.get_last_error()
== 2
    store.set("a", 1)
    success = success and store.get_last_error() == 0
    c = store.get("a")
    success = success and c == 1 and store.get_last_error() == 0
    store.set("b", 2)
    success = success and store.get_last_error() == 2
    store.add("x")
    store.input_var("x") # ввести значення x = 2
    success = success and store.get_last_error() == 0
    f = store.get("x")
    success = success and f == 2 and store.get_last_error() == 0
    store.clear()
    success = success and store.get_last_error() == 0
    store.add("a")
    success = success and store.get_last_error() == 0
    store.add("d")
    success = success and store.get_last_error() == 0
    store.input_all() # ввести значення a = 3, d = 4

```

```

    success = success and store.get_last_error() == 0
    c = store.get("a")
    success = success and c == 3 and store.get_last_error() == 0
    f = store.get("d")
    success = success and f == 4 and store.get_last_error() == 0
    success = success and store.is_in("a")
    success = success and {"a", "d"} ==
set(store.get_all().keys())

    print("Success =", success)

```

Головний модуль (повністю відправляється студентам разом з кістяком інших модулів).

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Модуль призначено для перевірки роботи модулів
tokenizer
syntax_analyzer
storage
та обчислення значення простого виразу (сума доданків)
"""

from tokenizer import get_tokens
from syntax_analyzer import SyntaxAnalyzer
from storage import Storage

def fill_storage(tokens, store):
    for tok in tokens:
        if tok.type == "variable":
            store.add(tok.value)

def compute_sum(tokens, store):
    s = 0
    for tok in tokens:
        if tok.type == "constant":
            value = float(tok.value)
            s += value
        elif tok.type == "variable":
            value = store.get(tok.value)
            s += value
    return s

expression = "5 + a + 3"

```

```

tokens = get_tokens(expression)

analyzer = SyntaxAnalyzer(tokens)
success, error = analyzer.check_expression_syntax()

if success:
    store = Storage()
    fill_storage(tokens, store)
    store.set("a", 2)
    s = compute_sum(tokens, store)
    success = s == 10

print("Success =", success)

```

Кістяк модуля tokenizer з глобальними константами та реалізованою функцією `_get_variable`.

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Модуль призначено для синтаксичного розбору виразу по частинах.

```

Вираз може мати вигляд:

$(abc + 123.5) * d2 - 3 / (x + y)$

Вираз може містити:

змінні - ідентифікатори  
 константи - дійсні або цілі числа без знаку  
 знаки операцій: +, -, \*, /  
 дужки: (, )

Функція `get_tokens` за заданим виразом має повертати послідовність лексем - токенів

Кожний токен - це кортеж:  $(\langle \text{тип токenu} \rangle, \langle \text{значення токenu} \rangle)$

```

from collections import namedtuple

# типи токенів
TOKEN_TYPE = ("variable",
              "constant",
              "operation",
              "left_paren",
              "right_paren",
              "other")

# словник фіксованих токенів, що складаються з одного символу
TOKEN_TYPES = {"+": "operation",
               "-": "operation",
               "*": "operation",

```

```

        "/": "operation",
        "(": "left_paren",
        ")": "right_paren"}

# тип токена
Token = namedtuple('Token', ['type', 'value'])

def get_tokens(string):
    """
    Функція за рядком повертає список tokenів типу Token
    :param string: рядок
    :return: список tokenів
    """

def _get_next_token(string):
    """
    Функція повертає наступний token та залишок рядка
    Використовує внутрішні функції _get_constant, _get_variable
    :param string: рядок
    :return: next_tok - наступний token, якщо є, або None
    :return: string - залишок рядка
    """

def _get_constant(string):
    """
    Функція за рядком повертає константу (якщо є) та залишок
    рядка
    :param string: рядок
    :return: константа (або порожній рядок), залишок рядка
    """

def _get_variable(string):
    """
    Функція за рядком повертає змінну (якщо є) та залишок рядка
    :param string: рядок
    :return: змінна (або порожній рядок), залишок рядка
    """
    if not string or not string[0].isalpha() and string[0] !=
    '_':
        return "", string

    for i, c in enumerate(string):
        if not c.isalnum() and c != '_':
            break

```

```

else:
    i += 1

return string[:i], string[i:]

if __name__ == "__main__":
    success = get_tokens("((ab1_ - 345.56) (* /.2{ _cde23") == (
        [Token(type='left_paren', value='('),
         Token(type='left_paren', value='('),
         Token(type='left_paren', value='('),
         Token(type='variable', value='ab1_'),
         Token(type='operation', value='-'),
         Token(type='constant', value='345.56'),
         Token(type='right_paren', value=')'),
         Token(type='left_paren', value='('),
         Token(type='operation', value='*'),
         Token(type='operation', value='/'),
         Token(type='other', value='.'),
         Token(type='constant', value='2'),
         Token(type='other', value='{'),
         Token(type='variable', value='_cde23')]
    print("Success =", success)

```

Кістяк модуля `syntax_analyzer` з глобальними константами та реалізованим конструктором класу `SyntaxAnalyzer`.

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
Модуль призначено для перевірки синтаксичної правильності виразу
та присвоєння.

```

Вираз може мати вигляд:

`(abc + 123.5)*d2-3/(x+y)`

Вираз може містити:

змінні - ідентифікатори

константи - дійсні або цілі числа без знаку

знаки операцій: +, -, \*, /

дужки: (, )

Присвоєння - це

`<змінна> = <вираз>`

наприклад

`x = a + b`



Метод `check_expression_syntax` за заданим списком токенів для виразу має повернути булівське значення та (можливо) помилку. Кожний токен – це кортеж: (`<тип токена>`, `<значення токена>`). Перевірка робиться на допустимість сусідніх токенів, правильний перший та останній токен, порожній вираз, правильність розставлення дужок.

Метод `check_assignment_syntax` за заданим списком токенів для присвоєння має повернути булівське значення та (можливо) помилку.

```
"""
from tokenizer import Token, get_tokens

# Словник множин допустимих наступних токенів для заданого токена
VALID_PAIRS = {"variable": {"operation", "right_paren"},
               "constant": {"operation", "right_paren"},
               "operation": {"variable", "constant",
                             "left_paren"},
               "left_paren": {"left_paren", "variable",
                              "constant"},
               "right_paren": {"right_paren", "operation"},
               "other": set()}

# Словник помилок
ERRORS = {"invalid_pair": "Недопустима пара токенів {}, {} ",
          "incorrect_parens": "Неправильно розставлені дужки",
          "empty_expr": "Порожній вираз"}

class SyntaxAnalyzer:
    def __init__(self, tokens):
        self._tokens = tokens[:]  # список токенів

    def check_expression_syntax(self):
        """
        Метод перевіряє синтаксичну правильність виразу за
        списком токенів. Використовує внутрішні методи _check_parens, _check_pair
        Повертає булівське значення та рядок помилки.
        Якщо помилки немає, то повертає порожній рядок
        :param tokens: список токенів
        :return: success – булівське значення
        :return: error – рядок помилки
        """
```

```

def _check_parens(self):
    """
    Метод перевіряє чи правильно розставлені дужки у виразі.
    Повертає булівське значення
    :param tokens: список токенів
    :return: success - булівське значення
    """

def _check_pair(self, tok, next_tok):
    """
    Метод перевіряє чи правильна пара токенів.
    Повертає булівське значення
    :param tok: поточний токен
    :param next_tok: наступний токен
    :return: success - булівське значення
    """

if __name__ == "__main__":
    analyzer = SyntaxAnalyzer(get_tokens("((ab1_ -
345.56) (* /.2{_cde23"))
    success1, error1 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzer(get_tokens("(ab1_ -
345.56) (* /.2_cde23"))
    success2, error2 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzer(get_tokens(" - 345.56* /.2_cde23"))
    success3, error3 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzer(get_tokens("2 - 345.56 *"))
    success4, error4 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzer(get_tokens("2 - .2"))
    success5, error5 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzer(get_tokens(" "))
    success6, error6 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzer(get_tokens("((abc -3 * b2) + d5 /
7)"))
    success7, error7 = analyzer.check_expression_syntax()

    success = (
        not success1 and error1 == 'Неправильно розставлені
дужки' and
        not success2 and error2 ==
        "Недопустима пара токенів Token(type='operation',
value='*'),"
        " Token(type='operation', value='/') and
        not success3 and not success4 and
        not success5 and error5 ==

```

```

        "Недопустима пара токенів Token(type='operation',
value='-'), "
        "Token(type='other', value='.')\" and
        not success6 and error6 == "Порожній вираз" and
        success7 and error7 == ""
    )

    print("Success =", success)

```

Кістяк модуля storage з глобальними константами, реалізованим конструктором та методом set класу Storage.

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
Модуль призначено для реалізації пам'яті, що складається зі
змінних.

Змінні можуть мати числові значення цілого або дійсного типу

"""

# словник, що співствляє коди помилок до їх описи
ERRORS = {0: "",
          1: "Змінна вже є у пам'яті",
          2: "Змінна не існує",
          3: "Змінна невизначена"}

class Storage():
    def __init__(self):
        self._storage = {}          # пам'ять
        self._last_error = 0        # код помилки останньої операції

    def add(self, variable):
        """
        Метод додає змінну у пам'ять.
        Якщо така змінна вже існує, то встановлює помилку
        :param variable: змінна
        :return: None
        """

    def is_in(self, variable):
        """
        Метод перевіряє, чи є змінна у пам'яті.

```

```

        :param variable: змінна
        :return: булівське значення (True, якщо є)
        """

def get(self, variable):
    """
    Метод повертає значення змінної.
    Якщо така змінна не існує або невизначена (==None),
    то встановлює відповідну помилку
    :param variable: змінна
    :return: значення змінної
    """

def set(self, variable, value):
    """
    Метод встановлює значення змінної
    Якщо змінна не існує, повертає помилку
    :param variable: змінна
    :param value: нове значення
    :return: None
    """

    if variable not in self._storage:
        self._last_error = 2
    else:
        self._last_error = 0
        self._storage[variable] = value

def input_var(self, variable):
    """
    Метод здійснює введення з клавіатури та встановлення
    значення змінної
    Якщо змінна не існує, повертає помилку
    :param variable: змінна
    :return: None
    """

def input_all(self):
    """
    Метод здійснює введення з клавіатури та встановлення
    значення
    усіх змінних з пам'яті
    :return: None
    """

```

```

def clear(self):
    """
    Метод видаляє усі змінні з пам'яті
    :return: None
    """

def get_last_error(self):
    """
    Метод повертає код останньої помилки code
    Для виведення повідомлення треба взяти
    storage.ERRORS[code]
    усіх змінних з пам'яті
    :return: код останньої помилки
    """

def get_all(self):
    """
    Метод повертає словник змінних пам'яті
    :return: словник змінних
    """

if __name__ == "__main__":
    store = Storage()
    store.add("a")
    success = store.get_last_error() == 0
    store.add("a")
    success = success and store.get_last_error() == 1
    c = store.get("a")
    success = success and c == None and store.get_last_error()
== 3
    c = store.get("b")
    success = success and c == None and store.get_last_error()
== 2
    store.set("a", 1)
    success = success and store.get_last_error() == 0
    c = store.get("a")
    success = success and c == 1 and store.get_last_error() == 0
    store.set("b", 2)
    success = success and store.get_last_error() == 2
    store.add("x")
    store.input_var("x") # ввести значення x = 2
    success = success and store.get_last_error() == 0
    f = store.get("x")

```

```

success = success and f == 2 and store.get_last_error() == 0
store.clear()
success = success and store.get_last_error() == 0
store.add("a")
success = success and store.get_last_error() == 0
store.add("d")
success = success and store.get_last_error() == 0
store.input_all() # ввести значення a = 3, d = 4
success = success and store.get_last_error() == 0
c = store.get("a")
success = success and c == 3 and store.get_last_error() == 0
f = store.get("d")
success = success and f == 4 and store.get_last_error() == 0
success = success and store.is_in("a")
success = success and {"a", "d"} ==
set(store.get_all().keys())

print("Success =", success)

```

## Групове завдання 16. Інтерпретатор лінійних програм. Крок 2

### Текст завдання

Треба розробити (додати) модулі `tokenizer`, `syntax_analyzer_ext`, `code_generator` згідно специфікації.

Це завдання продовжує завдання 15.

Треба

1. У модулі `tokenizer` забезпечити повернення токена для символу '=' (тип "equal") та перевірити правильність виконання.
2. Реалізувати модуль `syntax_analyzer_ext`, де описати клас `SyntaxAnalyzerExt` як нащадок `SyntaxAnalyzer`, у ньому описати метод `check_assignment_syntax` для перевірки синтаксису присвоєння.
3. Модуль `storage` та клас `Storage` не змінюються.
4. Реалізувати модуль `code_generator` та клас `CodeGenerator` для генерації низькорівневого програмного коду для обчислення виразів та реалізації присвоєнь.

Синтаксична діаграма виразу, близька до нашої задачі, зображена на рисунку нижче. Зокрема, у нас по-іншому позначаються змінні та константи (що не впливає на побудову коду). Все інше – відповідає.

Код програми розраховано на виконання з використанням стеку. Стек – це список, до якого ми можемо додавати елементи у кінець та забирати елементи з кінця. У стеку будуть зберігатись числа: константи та значення змінних, які вказано у виразі. Тому ці числа потрібно завантажити (додати) до стеку.

Для завантаження існують команди

*("LOADC", <число>) - завантажити число у стек  
("LOADV", <змінна>) - завантажити значення змінної у стек  
(використовується storage)*

Арифметичні дії будуть виконуватись командами

*("ADD", None) - обчислити суму двох верхніх елементів стеку  
("SUB", None) - обчислити різницю двох верхніх елементів стеку  
("MUL", None) - обчислити добуток двох верхніх елементів стеку  
("DIV", None) - обчислити частку від ділення двох верхніх елементів стеку*

Результат кожної арифметичної дії має завантажуватись у стек.

Присвоєння буде виконуватись командою

*("SET", <змінна>) - встановити (присвоїти) значення змінної у пам'яті (storage) рівним значенню останнього елементу стеку*

При присвоєнні елемент забирається зі стеку

Специфікація для пришвидшення розробки надається у вигляді «кістяка» модулів у файлах .ру у окремому архіві.

Після розробки кожного модуля його треба запустити окремо та досягти виведення значення

Success = True

Основну частину кожного модуля після

**if** \_\_name\_\_ == "\_\_main\_\_":

не змінювати (можна вставляти print для налагодження)

Усі функції модуля мають бути реалізовані

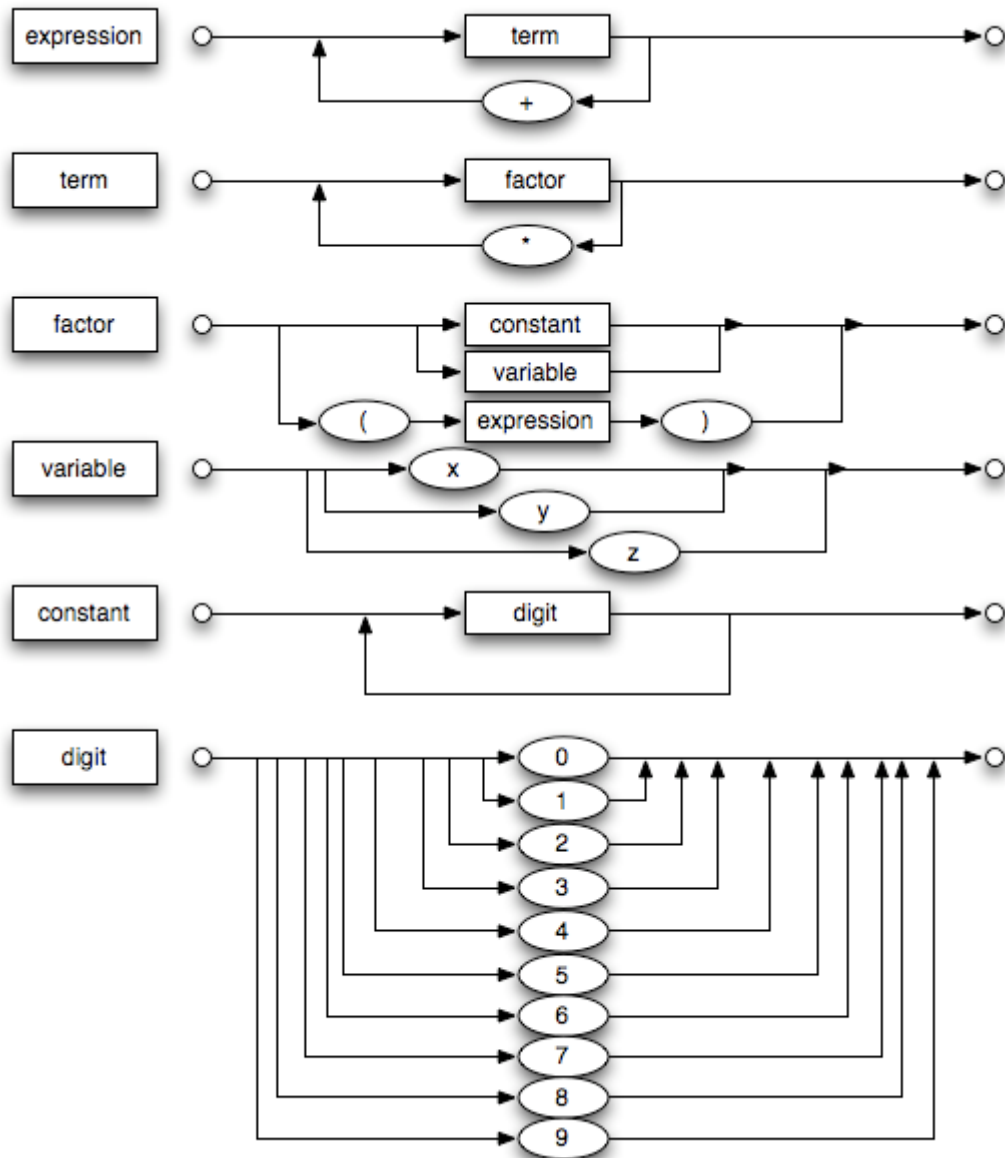
Заголовки функцій не змінювати

Можна додавати власні внутрішні функції у модулі, якщо потрібно.

Після завершення розробки модулів запустити модуль main та впевнитись, що виводиться

Success = True

Для неповних команд (з 2 або одного студента) достатньо реалізувати частину модулів по порядку (syntax\_analyzer\_ext, code\_generator)



### Необхідні передумови

Засвоєння Теми 13 «Класи та об'єкти».

### Мета завдання

Навчитись писати програми за заданою специфікацією у Python.

### Обмеження на виконання завдання

Немає.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо писати код модулів tokenizer, syntax\_analyzer\_ext, code\_generator.



## Зауваження щодо вирішення завдання

Це завдання є другим з 3 послідовних завдань, мета яких – написати власний інтерпретатор, який дозволить виконувати прості лінійні програми у мові, подібній до Python.

Перед заняттям студентам розсилається «кістяк» модулів, які треба наповнити програмним кодом. У кожному модулі є документація щодо розроблюваної функціональності. Кістяк модулів для даного заняття наведений після повного тексту програми. Також можуть розсилатись модулі з попереднього завдання (syntax\_analyzer та/або storage), якщо вони не були реалізовані жодною командою на попередньому занятті.

Також студенти повинні мати можливість після заняття і до наступного заняття обмінюватись працюючими модулями, які знадобляться для наступного заняття.

## Рекомендації щодо перевірки результатів

Перевірити правильність виконання тестів окремих модулів та головного модуля.

## Текст програми з розв'язками

Текст модуля . tokenizer у фінальному вигляді міститься у попередньому завданні

Модуль syntax\_analyzer\_ext

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
```

*Модуль призначено для перевірки синтаксичної правильності виразу та присвоєння.*

*Вираз може мати вигляд:*

*(abc + 123.5)\*d2-3/(x+y)*

*Вираз може містити:*

*змінні - ідентифікатори*

*константи - дійсні або цілі числа без знаку*

*знаки операцій: +, -, \*, /*

*дужки: (, )*

*Присвоєння - це*

*<змінна> = <вираз>*

*наприклад*

*x = a + b*

*Метод check\_assignment\_syntax за заданим списком токенів*

*для присвоєння має повернути*

*булівське значення та (можливо) помилку.*

```
"""
```

```
from tokenizer import Token, get_tokens
from syntax_analyzer import SyntaxAnalyzer
```

```

# СЛОВНИК ПОМИЛОК
ERRORS = {"invalid_pair": "Недопустима пара токенів {}, {}",
          "incorrect_parens": "Неправильно розставлені дужки",
          "empty_expr": "Порожній вираз",
          "incorrect_assignment": "Неправильне присвоєння"}

class SyntaxAnalyzerExt(SyntaxAnalyzer):
    def __init__(self, tokens):
        SyntaxAnalyzer.__init__(self, tokens)

    def check_assignment_syntax(self):
        """
        Метод перевіряє синтаксичну правильність присвоєння за
        списком токенів.
        Повертає булівське значення та рядок помилки.
        Якщо помилки немає, то повертає порожній рядок.
        Використовує метод check_expression_syntax
        :param tokens: список токенів
        :return: success - булівське значення
        :return: error - рядок помилки
        """
        if len(self._tokens) < 2 or \
            self._tokens[0].type != "variable" or \
            self._tokens[1].type != "equal":
            return False, ERRORS["incorrect_assignment"]

        tokens_copy = self._tokens[:]
        self._tokens = self._tokens[2:]
        result = self.check_expression_syntax()
        self._tokens = tokens_copy
        return result

if __name__ == "__main__":
    analyzer = SyntaxAnalyzerExt(get_tokens("((ab1_ -
345.56) (*.2{_cde23"))
    success1, error1 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("(ab1_ -
345.56) (*.2_cde23"))
    success2, error2 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens(" -
345.56 (*.2_cde23"))
    success3, error3 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("2 - 345.56 *"))
    success4, error4 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("2 - .2"))

```

```

    success5, error5 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens(" "))
    success6, error6 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("(abc -3 * b2) + d5
/ 7)"))
    success7, error7 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("x + y"))
    success8, error8 = analyzer.check_assignment_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("x ="))
    success9, error9 = analyzer.check_assignment_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("x = (a+b)"))
    success10, error10 = analyzer.check_assignment_syntax()

    success = (
        not success1 and error1 == 'Неправильно розставлені
дужки' and
        not success2 and error2 ==
            "Недопустима пара токенів Token(type='operation',
value='*'), "
            " Token(type='operation', value='/') and
        not success3 and not success4 and
        not success5 and error5 ==
            "Недопустима пара токенів Token(type='operation',
value='-'), "
            "Token(type='other', value='.') and
        not success6 and error6 == "Порожній вираз" and
        success7 and error7 == "" and
        not success8 and error8 == "Неправильне присвоєння" and
        not success9 and error9 == "Порожній вираз" and
        success10 and error10 == ""
    )

    print("Success =", success)

```

Модуль code\_generator

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""

```

Модуль призначено для генерації коду за списком рядків програми.  
 Генератор коду повертає список команд.  
 Кожна команда – це кортеж: (<код\_команди>, <операнд>)

У подальшому обчислення будуть виконуватись з використанням стеку.

Стек – це список, у який ми можемо додавати до кінця та брати з кінця числа.



```

class CodeGenerator:
    def __init__(self, program_lines, storage):
        self._storage = storage
        self._program_lines = program_lines

    def generate_code(self):
        """
        Метод генерує код за списком рядків програми
        program_lines
        Повертає програмний код у вигляді списку кортежів
        (<код_команди>, <операнд>)
        Також, якщо під час генерації коду або аналізу виникає
        помилка,
        то повертає текст помилки. Якщо помилки немає, то
        повертає порожній рядок.
        Побічний ефект: очищує пам'ять.
        :param program_lines: список рядків програми
        :return: список команд - кортежів (<код_команди>,
        <операнд>)
        :return: текст помилки
        """
        code = []
        self._storage.clear()
        for program_line in self._program_lines:
            line_code, error =
self._generate_line_code(program_line)
            if error:
                break
            code += line_code
        return code, error

    def _generate_line_code(self, program_line):
        """
        Метод генерує код за рядком програми program_line.
        Рядок програми має бути присвоєнням виду x = e,
        де x - змінна, e - вираз, або порожнім рядком.
        Використовує модулі tokenizer та syntax_analyzer для
        розбору
        та аналізу правильності синтаксису рядка програми.
        Використовує функцію _expression для генерації коду
        виразу, після чого
        генерує команду SET для змінної з лівої частини
        присвоєння, та додає
        змінну до пам'яті (storage), якщо потрібно.
        Якщо program_line - порожній рядок, то функція його
        ігнорує.

```

Повертає програмний код для рядка програми у вигляді списку кортежів  
 (<код\_команди>, <операнд>)  
 Також, якщо під час генерації коду або аналізу виникає помилка,  
 то повертає текст помилки. Якщо помилки немає, то повертає порожній рядок.

```

:param program_line: рядок програми
:return: список команд - кортежів (<код_команди>,
<операнд>)
:return: текст помилки
"""
code = []
tokens = get_tokens(program_line)
if not tokens:
    return code, ""

analyzer = SyntaxAnalyzerExt(tokens)
success, error = analyzer.check_assignment_syntax()
if error:
    return code, error

self.expression(code, tokens[2:])
variable = tokens[0].value
if not self._storage.is_in(variable):
    self._storage.add(variable)
code.append(("SET", variable))
return code, error

def _expression(self, code, tokens):
    """
    Метод генерує код за списком токенів виразу.
    Використовує функцію _term для генерації коду доданку,
    після чого,
    поки список токенів не спорожніє і поточний токен - це
    операція
    '+' або '-', знову використовує _term для наступного
    доданку та
    генерує команду ADD або SUB.
    Побічний ефект: змінює список code та список tokens
    (видаляє розглянуті токени)
    :param code: список команд - кортежів (<код_команди>,
    <операнд>)
    :param tokens: список токенів
    :return: None
    """
    self._term(code, tokens)

```

```

while tokens and tokens[0].type == "operation" \
    and tokens[0].value in ('+', '-'):
    token = tokens.pop(0)
    operation = token.value
    self._term(code, tokens)
    if operation == '+':
        code.append(("ADD", None))
    elif operation == '-':
        code.append(("SUB", None))

def _term(self, code, tokens):
    """
    Метод генерує код за списком токенів, що починається
    токенами доданку.
    Використовує функцію _factor для генерації коду
    множника, після чого,
    поки список токенів не спорожніє і поточний токен - це
    операція
    '*' або '/', знову використовує _factor для наступного
    множника та
    генерує команду MUL або DIV.
    Побічний ефект: змінює список code та список tokens
    (видаляє розглянуті токени)
    :param code: список команд - кортежів (<код_команди>,
    <операнд>)
    :param tokens: список токенів
    :return: None
    """
    self._factor(code, tokens)
    while tokens and tokens[0].type == "operation" \
        and tokens[0].value in ('*', '/'):
        token = tokens.pop(0)
        operation = token.value
        self._factor(code, tokens)
        if operation == '*':
            code.append(("MUL", None))
        elif operation == '/':
            code.append(("DIV", None))

def _factor(self, code, tokens):
    """
    Метод генерує код за списком токенів, що починається
    токенами множника.
    Якщо перший токен - "left_paren", то множник - це вираз
    у дужках і треба
    викликати функцію _expression, після чого пропустити

```

праву дужку.

Якщо перший токен - константа або змінна, то треба згенерувати команду

LOADC (додатково - перетворити константу з рядка у дійсне число) або

LOADV (додатково - додати змінну до пам'яті, якщо необхідно).

Побічний ефект: змінює список code та список tokens (видаляє розглянуті токени)

:param code: список команд - кортежів (<код\_команди>, <операнд>)

:param tokens: список токенів

:return: None

"""

token = tokens.pop(0)

if token.type == "left\_paren":

self.\_expression(code, tokens)

if not tokens or tokens[0].type != "right\_paren":

print("Factor: ')' expected")

return

tokens.pop(0)

elif token.type == "constant":

code.append(("LOADC", float(token.value)))

elif token.type == "variable":

variable = token.value

if not self.\_storage.is\_in(variable):

self.\_storage.add(variable)

code.append(("LOADV", variable))

else:

print("Factor: Invalid token", token.type)

def in\_storage(self, variable):

"""

Метод перевіряє, чи міститься змінна variable у пам'яті.

:param variable: ім'я змінної

:return: bool - чи міститься змінна variable у пам'яті

"""

return self.\_storage.is\_in(variable)

if \_\_name\_\_ == "\_\_main\_\_":

generator = CodeGenerator(["a = b + c", "y = (2 - 1]",  
Storage())

code, error = generator.generate\_code()

success = error == "Неправильно розставлені дужки"

generator = CodeGenerator(["x = 1",  
"z = ((a))",



```

                "a = b + c * (d - e)",
                "y = (2 - 1) * (x345 + 3 * d) /
234.5 - z"],

                Storage())
code, error = generator.generate_code()
success = success and not error and \
    code == [('LOADC', 1.0),
              ('SET', 'x'),
              ('LOADV', 'a'),
              ('SET', 'z'),
              ('LOADV', 'b'),
              ('LOADV', 'c'),
              ('LOADV', 'd'),
              ('LOADV', 'e'),
              ('SUB', None),
              ('MUL', None),
              ('ADD', None),
              ('SET', 'a'),
              ('LOADC', 2.0),
              ('LOADC', 1.0),
              ('SUB', None),
              ('LOADV', 'x345'),
              ('LOADC', 3.0),
              ('LOADV', 'd'),
              ('MUL', None),
              ('ADD', None),
              ('MUL', None),
              ('LOADC', 234.5),
              ('DIV', None),
              ('LOADV', 'z'),
              ('SUB', None),
              ('SET', 'y')]

success = success and generator.in_storage('a')
success = success and generator.in_storage('x')

print("Success =", success)

```

Кістяк модуля tokenizer фактично повторює попереднє завдання окремі глобальних констант:

```

# типи токенів
TOKEN_TYPE = ("variable",
               "constant",
               "operation",
               "left_paren",
               "right_paren",
               "other",

```

```
"equal")
```

```
# словник фіксованих токенів, що складаються з одного символу
TOKEN_TYPES = {"+": "operation",
               "-": "operation",
               "*": "operation",
               "/": "operation",
               "(": "left_paren",
               ")": "right_paren",
               "=": "equal"}
```

Кістяк модуля `syntax_analyzer_ext` з реалізованим конструктором класу `SyntaxAnalyzerExt`.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
```

Модуль призначено для перевірки синтаксичної правильності виразу та присвоєння.

Вираз може мати вигляд:

$(abc + 123.5) * d2 - 3 / (x + y)$

Вираз може містити:

змінні - ідентифікатори

константи - дійсні або цілі числа без знаку

знаки операцій:  $+$ ,  $-$ ,  $*$ ,  $/$

дужки:  $(, )$

Присвоєння - це

$\langle \text{змінна} \rangle = \langle \text{вираз} \rangle$

наприклад

$x = a + b$

Метод `check_assignment_syntax` за заданим списком токенів для присвоєння має повернути булівське значення та (можливо) помилку.

```
from tokenizer import Token, get_tokens
from syntax_analyzer import SyntaxAnalyzer
```

```
# словник помилок
```

```
ERRORS = {"invalid_pair": "Недопустима пара токенів {}, {}",
          "incorrect_parens": "Неправильно розставлені дужки",
          "empty_expr": "Порожній вираз",
          "incorrect_assignment": "Неправильне присвоєння"}
```

```

class SyntaxAnalyzerExt(SyntaxAnalyzer):
    def __init__(self, tokens):
        SyntaxAnalyzer.__init__(self, tokens)

    def check_assignment_syntax(self):
        """
        Метод перевіряє синтаксичну правильність присвоєння за
        списком токенів.
        Повертає булівське значення та рядок помилки.
        Якщо помилки немає, то повертає порожній рядок.
        Використовує метод check_expression_syntax
        :param tokens: список токенів
        :return: success - булівське значення
        :return: error - рядок помилки
        """

if __name__ == "__main__":
    analyzer = SyntaxAnalyzerExt(get_tokens("((ab1_ -
345.56) (* /.2{_cde23"))
    success1, error1 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("(ab1_ -
345.56) (* /.2_cde23"))
    success2, error2 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens(" -
345.56* /.2_cde23"))
    success3, error3 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("2 - 345.56 *"))
    success4, error4 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("2 - .2"))
    success5, error5 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens(" "))
    success6, error6 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("(abc -3 * b2) + d5
/ 7)"))
    success7, error7 = analyzer.check_expression_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("x + y"))
    success8, error8 = analyzer.check_assignment_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("x ="))
    success9, error9 = analyzer.check_assignment_syntax()
    analyzer = SyntaxAnalyzerExt(get_tokens("x = (a+b)"))
    success10, error10 = analyzer.check_assignment_syntax()

    success = (
        not success1 and error1 == 'Неправильно розставлені
дужки' and

```

```

        not success2 and error2 ==
        "Недопустима пара токенів Token(type='operation',
value='*'), "
        " Token(type='operation', value='/') " and
        not success3 and not success4 and
        not success5 and error5 ==
        "Недопустима пара токенів Token(type='operation',
value='-'), "
        "Token(type='other', value='.') " and
        not success6 and error6 == "Порожній вираз" and
        success7 and error7 == "" and
        not success8 and error8 == "Неправильне присвоєння" and
        not success9 and error9 == "Порожній вираз" and
        success10 and error10 == ""
    )

    print("Success =", success)

```

Кістяк модуля code\_generator з реалізованим конструктором класу CodeGenerator та методами \_term та in\_storage.

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""

```

Модуль призначено для генерації коду за списком рядків програми. Генератор коду повертає список команд.

Кожна команда – це кортеж: (<код\_команди>, <операнд>)

У подальшому обчислення будуть виконуватись з використанням стеку.

Стек – це список, у який ми можемо додавати до кінця та брати з кінця числа.

Для виконання арифметичної операції буде братись два останніх числа зі стеку, обчислювати результат операції та додавати результат до стеку. Тому генератор повинен згенерувати команди завантаження змінних та констант до стеку а також виконання арифметичних операцій та присвоєння.

Допустимі команди:

("LOADC", <число>) – завантажити число у стек

("LOADV", <змінна>) – завантажити значення змінної у стек  
(використовується storage)

("ADD", None) – обчислити суму двох верхніх елементів стеку

("SUB", None) – обчислити різницю двох верхніх елементів стеку

("MUL", None) – обчислити добуток двох верхніх елементів стеку

("DIV", None) - обчислити частку від ділення двох верхніх елементів стеку  
("SET", <змінна>) - встановити (присвоїти) значення змінної у пам'яті (storage) рівним значенню останнього елементу стеку

Генерація коду виконується за допомогою рекурсивного розбору виразу.

Вираз (expression) представляється як один доданок (term) або сума (різниця)

багатьох доданків.

Доданок (term) представляється як один множник (factor) або добуток

(частка від ділення) багатьох множників.

Множник (factor) представляється як константа або змінна, або вираз (expression) у дужках.

Під час розбору кожен метод забирає токени зі списку tokenів tokens,

а також додає команди до списку команд code

"""

```
from storage import Storage
from tokenizer import get_tokens
from syntax_analyzer_ext import SyntaxAnalyzerExt
```

```
COMMANDS = ("LOADC",
            "LOADV",
            "ADD",
            "SUB",
            "MUL",
            "DIV",
            "SET")
```

```
class CodeGenerator:
```

```
    def __init__(self, program_lines, storage):
        self._storage = storage
        self._program_lines = program_lines
```

```
    def generate_code(self):
        """
```

Метод генерує код за списком рядків програми  
program\_lines  
Повертає програмний код у вигляді списку кортежів  
(<код\_команди>, <операнд>)  
Також, якщо під час генерації коду або аналізу виникає помилка,  
то повертає текст помилки. Якщо помилки немає, то

повертає порожній рядок.

Побічний ефект: очищує пам'ять.

**:param** program\_lines: список рядків програми

**:return:** список команд - кортежів (<код\_команди>,

<операнд>)

**:return:** текст помилки

"""

**def** \_generate\_line\_code(self, program\_line):

"""

Метод генерує код за рядком програми program\_line.

Рядок програми має бути присвоєнням виду x = e,

де x - змінна, e - вираз, або порожнім рядком.

Використовує модулі tokenizer та syntax\_analyzer для

розбору

та аналізу правильності синтаксису рядка програми.

Використовує функцію \_expression для генерації коду

виразу, після чого

генерує команду SET для змінної з лівої частини

присвоєння, та додає

змінну до пам'яті (storage), якщо потрібно.

Якщо program\_line - порожній рядок, то функція його

ігнорує.

Повертає програмний код для рядка програми у вигляді

списку кортежів

(<код\_команди>, <операнд>)

Також, якщо під час генерації коду або аналізу виникає

помилка,

то повертає текст помилки. Якщо помилки немає, то

повертає порожній рядок.

**:param** program\_line: рядок програми

**:return:** список команд - кортежів (<код\_команди>,

<операнд>)

**:return:** текст помилки

"""

**def** \_expression(self, code, tokens):

"""

Метод генерує код за списком токенів виразу.

Використовує функцію \_term для генерації коду доданку,

після чого,

поки список токенів не спорожніє і поточний токен - це

операція

'+' або '-', знову використовує \_term для наступного

доданку та

```

генерує команду ADD або SUB.
Побічний ефект: змінює список code та список tokens
(видаляє розглянуті токени)
:param code: список команд - кортежів (<код_команди>,
<операнд>)
:param tokens: список токенів
:return: None
"""

def _term(self, code, tokens):
    """
    Метод генерує код за списком токенів, що починається
    токенами доданку.
    Використовує функцію _factor для генерації коду
    множника, після чого,
    поки список токенів не спорожніє і поточний токен - це
    операція
    '*' або '/', знову використовує _factor для наступного
    множника та
    генерує команду MUL або DIV.
    Побічний ефект: змінює список code та список tokens
    (видаляє розглянуті токени)
    :param code: список команд - кортежів (<код_команди>,
    <операнд>)
    :param tokens: список токенів
    :return: None
    """
    self._factor(code, tokens)
    while tokens and tokens[0].type == "operation" \
        and tokens[0].value in ('*', '/'):
        token = tokens.pop(0)
        operation = token.value
        self._factor(code, tokens)
        if operation == '*':
            code.append(("MUL", None))
        elif operation == '/':
            code.append(("DIV", None))

def _factor(self, code, tokens):
    """
    Метод генерує код за списком токенів, що починається
    токенами множника.
    Якщо перший токен - "left_paren", то множник - це вираз
    у дужках і треба
    викликати функцію _expression, після чого пропустити
    праву дужку.

```

Якщо перший токен - константа або змінна, то треба згенерувати команду

LOADC (додатково - перетворити константу з рядка у дійсне число) або

LOADV (додатково - додати змінну до пам'яті, якщо необхідно).

Побічний ефект: змінює список code та список tokens (видаляє розглянуті токени)

**:param** code: список команд - кортежів (<код\_команди>, <операнд>)

**:param** tokens: список токенів

**:return:** None

"""

```
def in_storage(self, variable):
    """
    Метод перевіряє, чи міститься змінна variable у пам'яті.
    :param variable: ім'я змінної
    :return: bool - чи міститься змінна variable у пам'яті
    """
    return self._storage.is_in(variable)
```

```
if __name__ == "__main__":
    generator = CodeGenerator(["a = b + c", "y = (2 - 1)",
                               Storage()])
    code, error = generator.generate_code()
    success = error == "Неправильно розставлені дужки"

    generator = CodeGenerator(["x = 1",
                               "z = ((a))",
                               "a = b + c * (d - e)",
                               "y = (2 - 1) * (x345 + 3 * d) /
234.5 - z"],
                               Storage())
    code, error = generator.generate_code()
    success = success and not error and \
        code == [('LOADC', 1.0),
                  ('SET', 'x'),
                  ('LOADV', 'a'),
                  ('SET', 'z'),
                  ('LOADV', 'b'),
                  ('LOADV', 'c'),
                  ('LOADV', 'd'),
                  ('LOADV', 'e'),
                  ('SUB', None),
                  ('MUL', None),
```



```
( 'ADD', None),
( 'SET', 'a'),
( 'LOADC', 2.0),
( 'LOADC', 1.0),
( 'SUB', None),
( 'LOADV', 'x345'),
( 'LOADC', 3.0),
( 'LOADV', 'd'),
( 'MUL', None),
( 'ADD', None),
( 'MUL', None),
( 'LOADC', 234.5),
( 'DIV', None),
( 'LOADV', 'z'),
( 'SUB', None),
( 'SET', 'y') ]
```

```
success = success and generator.in_storage('a')
success = success and generator.in_storage('x')
```

```
print("Success =", success)
```

## Групове завдання 17. Інтерпретатор лінійних програм. Крок 3

### Текст завдання

Треба розробити модуль interpreter та клас Interpreter.

Це завдання продовжує завдання 15, 16.

Треба

1. Реалізувати модуль interpreter та клас Interpreter для інтерпретації (виконання) низькорівневого програмного коду для обчислення виразів та реалізації присвоєнь. Інтерпретатор виконує лінійну програму, яка складається з присвоєнь.

Код програми розраховано на виконання з використанням стеку. Стек – це **список**, до якого ми можемо додавати елементи у кінець та забирати елементи з кінця. У стеку будуть зберігатись числа: константи та значення змінних, які вказано у виразі. Тому ці числа потрібно завантажити (додати) до стеку.

Для завантаження існують команди

```
("LOADC", <число>) - завантажити число у стек
("LOADV", <змінна>) - завантажити значення змінної у стек
(використовується storage)
```

Арифметичні дії будуть виконуватись командами

```
("ADD", None) - обчислити суму двох верхніх елементів стеку
("SUB", None) - обчислити різницю двох верхніх елементів стеку
```

*("MUL", None) - обчислити добуток двох верхніх елементів стеку  
("DIV", None) - обчислити частку від ділення двох верхніх елементів  
стеку*

Присвоєння буде виконуватись командою

*("SET", <змінна>) - встановити (присвоїти) значення змінної  
у пам'яті (storage) рівним  
значенню останнього елементу стеку*

Після реалізації модуля треба перевірити його правильність, виконавши модуль та отримавши результат

Success = True

Далі виконати головний модуль main2 та отримати результат Success = True. Додати свої програми та виконати їх у головному модулі

У архіві наведено повний текст модуля генератора коду (code\_generator). Можна скористатись цим модулем або взяти свій модуль, якщо він готовий.

Специфікація для пришвидшення розробки надається у вигляді «кістяка» модулів у файлах .py у окремому архіві.

Після розробки кожного модуля його треба запустити окремо та досягти виведення значення

Success = True

Основну частину кожного модуля після  
**if \_\_name\_\_ == "\_\_main\_\_":**  
не змінювати (можна вставляти print для налагодження)

Усі функції модуля мають бути реалізовані

Заголовки функцій не змінювати

Можна додавати власні внутрішні функції у модулі, якщо потрібно.

Після завершення розробки модулів запустити модуль main та впевнитись, що виводиться

Success = True

## Необхідні передумови

Засвоєння Теми 13 «Класи та об'єкти».

## Мета завдання

Навчитись писати програми за заданою специфікацією у Python.

## Обмеження на виконання завдання

Немає.

## Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо писати методи, які реалізують дії над стеком: SUB тощо.

## Зауваження щодо вирішення завдання

Це завдання є третім з 3 послідовних завдань, мета яких – написати власний інтерпретатор, який дозволить виконувати прості лінійні програми у мові, подібній до Python.

Перед заняттям студентам розсилається «кістяк» модулів, які треба наповнити програмним кодом. У кожному модулі є документація щодо розроблюваної функціональності. Кістяк модулів для даного заняття наведений після повного тексту програми. Також можуть розсилатись модулі з попереднього завдання (`syntax_analyzer_ext` та/або `code_generator`), якщо вони не були реалізовані жодною командою на попередньому занятті.

Окрім кістяка програм у Python, студентам також розсилаються приклади лінійних програм, які може виконувати інтерпретатор. Ці приклади використовуються у тестуванні головного модуля.

## Рекомендації щодо перевірки результатів

Перевірити правильність виконання тестів окремих модулів та головного модуля.

## Текст програми з розв'язками

Текст модуля `interpreter`

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
```

*Модуль призначено для виконання коду, який згенеровано генератором коду.*

*Генератор коду повертає список команд.*

*Кожна команда – це кортеж: (<код\_команди>, <операнд>)*

*Інтерпретатор виконує обчислення з використанням стеку.*

*Стек – це список, у який ми можемо додавати до кінця та брати з кінця числа.*

*Щоб додати число до стеку, можна використати*

*`_stack.append(number)`*

*Щоб взяти число зі стеку, можна використати*

*`number = _stack.pop()`*

*Для виконання арифметичної операції інтерпретатор бере два останніх числа зі стеку,*

*обчислює результат операції та додає результат до стеку.*

*Допустимі команди:*

*("LOADC", <число>) – завантажити число у стек*

*("LOADV", <змінна>) – завантажити значення змінної у стек (використовується `storage`)*

*("ADD", None) – обчислити суму двох верхніх елементів стеку*

*("SUB", None) – обчислити різницю двох верхніх елементів стеку*

*("MUL", None) – обчислити добуток двох верхніх елементів стеку*

*("DIV", None) – обчислити частку від ділення двох верхніх елементів стеку*

*("SET", <змінна>) – встановити значення змінної у пам'яті*

```

(storage)
"""
from storage import Storage

# СЛОВНИК, ЩО СПІВСТВЛЯЄ КОДИ ПОМИЛОК ДО ЇХ ОПИСИ
ERRORS = {0: "",
          1: "Недопустима команда",
          2: "Змінна не існує",
          3: "Ділення на 0"}

class Interpreter():
    def __init__(self, code, storage):
        self._code = code          # програмний код (результат
роботи                               # генератора коду)

        self._storage = storage    # пам'ять
        self._stack = []           # стек інтерпретатора
                                    # для виконання обчислень
        self._last_error = 0       # код помилки останньої
операції

        self._command_funcs = {    # словник методів обробки
КОМАНД
            "LOADC": self._loadc,
            "LOADV": self._loadv,
            "ADD": self._add,
            "SUB": self._sub,
            "MUL": self._mul,
            "DIV": self._div,
            "SET": self._set,
        }

    def _loadc(self, number):
        """
        Метод завантажує число у стек.
        Щоб додати у стек, використовує _stack.append(...)
        Побічний ефект: встановлює значення _last_error у 0
        :param number: число
        :return: None
        """
        self._last_error = 0
        self._stack.append(number)

    def _loadv(self, variable):
        """
        Метод завантажує значення змінної з пам'яті у стек.

```

```

        Якщо змінної не існує, то встановлює відповідну помилку.
        Якщо змінна не визначена, вводить значення змінної
        за допомогою storage.
        Використовує модуль storage
        Щоб додати у стек, використовує _stack.append(...)
        Побічний ефект: змінює значення _last_error
        :param variable: ім'я змінної
        :return: None
        """
        if not self._storage.is_in(variable):
            self._last_error = 2
            return

        self._last_error = 0
        value = self._storage.get(variable)
        if value is None:
            self._storage.input_var(variable)
            value = self._storage.get(variable)
        self._stack.append(value)

def _add(self, _=None):
    """
    Метод бере 2 останніх елемента зі стеку,
    обчислює їх суму та додає результат у стек.
    Щоб взяти значення зі стеку, використовує _stack.pop()
    Щоб додати у стек, використовує _stack.append(...)
    Побічний ефект: встановлює значення _last_error у 0
    :param _: ігнорується
    :return: None
    """
    self._last_error = 0
    second = self._stack.pop()
    first = self._stack.pop()
    self._stack.append(first + second)

def _sub(self, _=None):
    """
    Метод бере 2 останніх елемента зі стеку,
    обчислює їх різницю та додає результат у стек.
    Щоб взяти значення зі стеку, використовує _stack.pop()
    Щоб додати у стек, використовує _stack.append(...)
    Побічний ефект: встановлює значення _last_error у 0
    :param _: ігнорується
    :return: None
    """
    self._last_error = 0
    second = self._stack.pop()

```

```

first = self._stack.pop()
self._stack.append(first - second)

def _mul(self, _=None):
    """
    Метод бере 2 останніх елемента зі стеку,
    обчислює їх добуток та додає результат у стек.
    Щоб взяти значення зі стеку, використовує _stack.pop()
    Щоб додати у стек, використовує _stack.append(...)
    Побічний ефект: встановлює значення _last_error у 0
    :param _: ігнорується
    :return: None
    """
    self._last_error = 0
    second = self._stack.pop()
    first = self._stack.pop()
    self._stack.append(first * second)

def _div(self, _=None):
    """
    Метод бере останній та передостанній елементи зі стеку,
    обчислює частку від ділення передостаннього елемента на
    останній
    та додає результат у стек.
    Якщо дільник - 0, то встановлює помилку.
    Щоб взяти значення зі стеку, використовує _stack.pop()
    Щоб додати у стек, використовує _stack.append(...)
    Побічний ефект: встановлює значення _last_error у 0
    :param _: ігнорується
    :return: None
    """
    self._last_error = 0
    second = self._stack.pop()
    first = self._stack.pop()
    if second == 0:
        self._last_error = 3
        return

    self._stack.append(first / second)

def _set(self, variable):
    """
    Метод бере останній елемент зі стеку
    та встановлює значення змінної рівним цьому елементу.
    Якщо змінної не існує, то встановлює відповідну помилку.
    Щоб взяти значення зі стеку, використовує _stack.pop()
    Побічний ефект: змінює значення _last_error

```

```

:param variable: ім'я змінної
:return: None
"""
if not self._storage.is_in(variable):
    self._last_error = 2
    return

self._last_error = 0
value = self._stack.pop()
self._storage.set(variable, value)

def execute(self):
    """
    Метод виконує код програми, записаний у self._code.
    Повертає код останньої помилки або 0, якщо помилки
    немає.

    Якщо є помилка, то показує її.
    Використовує словник функцій COMMAND_FUNCS
    :return: код останньої помилки або 0, якщо помилки немає
    """
    for command in self._code:
        func = self._command_funcs.get(command[0])
        if not func:
            self._last_error = 1
        else:
            func(command[1])
            if self._last_error:
                print ("Помилка виконання:
{}").format(ERRORS[self._last_error]))
                break

    return self._last_error

def show_variables(self):
    """
    Метод показує значення усіх змінних пам'яті
    у форматі <змінна> = <значення>
    :return: None
    """
    variables = self._storage.get_all()
    for variable in variables:
        print("{} = {}".format(variable,
variables[variable]))

def get_value(self, variable):
    """
    Метод повертає значення змінної variable з пам'яті

```

*Якщо змінної у пам'яті немає або вона невизначена,  
повертає None*

```
:param variable: ім'я змінної
:return: float (or None)
"""
value = None
if self._storage.is_in(variable):
    value = self._storage.get(variable)
return value

def add_to_storage(self, variable):
    """
    Метод додає змінну variable до пам'яті
    :param variable: ім'я змінної
    :return: None
    """
    self._storage.add(variable)

if __name__ == "__main__":
    code = [('LOADC', 1.0),
            ('SET', 'x'),
            ('LOADC', 1.0),
            ('SET', 'y'),
            ('LOADV', 'x'),
            ('LOADV', 'a'),
            ('MUL', None),
            ('SET', 't'),
            ('LOADC', 1.0),
            ('LOADV', 'x'),
            ('LOADV', 'y'),
            ('SUB', None),
            ('DIV', None),
            ('SET', 'z')]
    interpreter = Interpreter(code, Storage())
    interpreter.add_to_storage('x')
    interpreter.add_to_storage('y')
    interpreter.add_to_storage('a')
    interpreter.add_to_storage('t')
    interpreter.add_to_storage('z')
    last_error = interpreter.execute()

    success = last_error == 3

    code = [('XXX', 1.0),
            ('SET', 'x'),
```



```

        ('LOADC', 1.0),
        ('SET', 'y')]
interpreter = Interpreter(code, Storage())
last_error = interpreter.execute()

success = success and last_error == 1

code = [ ('LOADC', 1.0),
        ('SET', 'x'),
        ('LOADC', 1.0),
        ('SET', 'y')]
interpreter = Interpreter(code, Storage())
last_error = interpreter.execute()

success = success and last_error == 2

code = [ ('LOADC', 2.0),
        ('SET', 'x'),
        ('LOADC', 1.0),
        ('SET', 'y'),
        ('LOADC', 1.0),
        ('LOADV', 'x'),
        ('LOADV', 'y'),
        ('SUB', None),
        ('DIV', None),
        ('SET', 'z')]
interpreter = Interpreter(code, Storage())
interpreter.add_to_storage('x')
interpreter.add_to_storage('y')
interpreter.add_to_storage('z')
last_error = interpreter.execute()

z = interpreter.get_value('z')
success = success and last_error == 0 and z == 1.0

print("Success =", success)

```

#### Головний модуль

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
Модуль призначено для перевірки роботи модулів
tokenizer
syntax_analyzer
storage

```

```

code_generator
interpreter
та виконання лінійних програм, що складаються з присвоєнь
"""

from storage import Storage
from code_generator import CodeGenerator
from interpreter import Interpreter, ERRORS

def load_program(filename):
    """
    Функція завантажує програму з файлу filename
    та повертає список рядків програми.
    :param filename: ім'я файлу
    :return: список рядків
    """
    f = open(filename, 'r')
    program_lines = f.read().splitlines()
    f.close()
    return program_lines

def print_program(program_lines):
    """
    Функція показує програму за списком її рядків
    :param program_lines: список рядків програми
    :return: None
    """
    for line in program_lines:
        print(line)

def execute_program(program_lines):
    """
    Функція виконує програму та показує стан пам'яті після
    виконання
    :param program_lines: список рядків програми
    :return: None
    """
    print_program(program_lines)

    storage = Storage()
    cd = CodeGenerator(program_lines, storage)
    code, error = cd.generate_code()
    if error:
        print("Помилка при генерації коду: {}".format(error))

```

```

        return None, error

    interpreter = Interpreter(code, storage)
    last_error = interpreter.execute()
    if last_error:
        error = ERRORS[last_error]
        print("Помилка виконання програми {}".format(error))
        return interpreter, error

    print("Стан пам'яті")
    interpreter.show_variables()
    return interpreter, ""

if __name__ == '__main__':
    print("\nprogram1")
    interpreter, error =
execute_program(load_program('program1.txt'))
    success = error == "Неправильно розставлені дужки"

    print("\nprogram2")
    interpreter, error =
execute_program(load_program('program2.txt'))
    success = success and error == "Ділення на 0"

    print("\nprogram3")
    interpreter, error =
execute_program(load_program('program3.txt'))
    z = interpreter.get_value('z')
    success = success and error == "" and z == 27.0

    print("\nSuccess =", success)

```

Кістяк модуля interpreter з реалізованим конструктором та методом `_add`.

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""

```

Модуль призначено для виконання коду, який згенеровано генератором коду.

Генератор коду повертає список команд.

Кожна команда – це кортеж: (<код\_команди>, <операнд>)

Інтерпретатор виконує обчислення з використанням стеку.

Стек – це список, у який ми можемо додавати до кінця та брати з кінця числа.

Щоб додати число до стеку, можна використати

```
_stack.append(number)
```

Щоб взяти число зі стеку, можна використати

```
number = _stack.pop()
```

Для виконання арифметичної операції інтерпретатор бере два останніх числа зі стеку, обчислює результат операції та додає результат до стеку.

Допустимі команди:

("LOADC", <число>) - завантажити число у стек

("LOADV", <змінна>) - завантажити значення змінної у стек

(використовується storage)

("ADD", None) - обчислити суму двох верхніх елементів стеку

("SUB", None) - обчислити різницю двох верхніх елементів стеку

("MUL", None) - обчислити добуток двох верхніх елементів стеку

("DIV", None) - обчислити частку від ділення двох верхніх елементів стеку

("SET", <змінна>) - встановити значення змінної у пам'яті (storage)

"""

```
from storage import Storage
```

# словник, що співствляє коди помилок до їх описи

```
ERRORS = {0: "",
          1: "Недопустима команда",
          2: "Змінна не існує",
          3: "Ділення на 0"}
```

```
class Interpreter():
```

```
    def __init__(self, code, storage):
```

```
        self._code = code          # програмний код (результат
роботи
```

```
        # генератора коду)
```

```
        self._storage = storage    # пам'ять
```

```
        self._stack = []          # стек інтерпретатора
```

```
        # для виконання обчислень
```

```
        self._last_error = 0       # код помилки останньої
операції
```

```
        self._command_funcs = {    # словник методів обробки
команд
```

```
            "LOADC": self._loadc,
```

```
            "LOADV": self._loadv,
```

```
            "ADD": self._add,
```

```
            "SUB": self._sub,
```

```
            "MUL": self._mul,
```

```
            "DIV": self._div,
```

```

        "SET": self._set,
    }

def _loadc(self, number):
    """
    Метод завантажує число у стек.
    Щоб додати у стек, використовує _stack.append(...)
    Побічний ефект: встановлює значення _last_error у 0
    :param number: число
    :return: None
    """

def _loadv(self, variable):
    """
    Метод завантажує значення змінної з пам'яті у стек.
    Якщо змінної не існує, то встановлює відповідну помилку.
    Якщо змінна не визначена, вводить значення змінної
    за допомогою storage.
    Використовує модуль storage
    Щоб додати у стек, використовує _stack.append(...)
    Побічний ефект: змінює значення _last_error
    :param variable: ім'я змінної
    :return: None
    """

def _add(self, _=None):
    """
    Метод бере 2 останніх елемента зі стеку,
    обчислює їх суму та додає результат у стек.
    Щоб взяти значення зі стеку, використовує _stack.pop()
    Щоб додати у стек, використовує _stack.append(...)
    Побічний ефект: встановлює значення _last_error у 0
    :param _: ігнорується
    :return: None
    """
    self._last_error = 0
    second = self._stack.pop()
    first = self._stack.pop()
    self._stack.append(first + second)

def _sub(self, _=None):
    """
    Метод бере 2 останніх елемента зі стеку,
    обчислює їх різницю та додає результат у стек.
    Щоб взяти значення зі стеку, використовує _stack.pop()

```

```

        Щоб додати у стек, використовує _stack.append(...)
        Побічний ефект: встановлює значення _last_error у 0
        :param _: ігнорується
        :return: None
        """

def _mul(self, _=None):
    """
    Метод бере 2 останніх елемента зі стеку,
    обчислює їх добуток та додає результат у стек.
    Щоб взяти значення зі стеку, використовує _stack.pop()
    Щоб додати у стек, використовує _stack.append(...)
    Побічний ефект: встановлює значення _last_error у 0
    :param _: ігнорується
    :return: None
    """

def _div(self, _=None):
    """
    Метод бере останній та передостанній елементи зі стеку,
    обчислює частку від ділення передостаннього елемента на
    останній
    та додає результат у стек.
    Якщо дільник - 0, то встановлює помилку.
    Щоб взяти значення зі стеку, використовує _stack.pop()
    Щоб додати у стек, використовує _stack.append(...)
    Побічний ефект: встановлює значення _last_error у 0
    :param _: ігнорується
    :return: None
    """

def _set(self, variable):
    """
    Метод бере останній елемент зі стеку
    та встановлює значення змінної рівним цьому елементу.
    Якщо змінної не існує, то встановлює відповідну помилку.
    Щоб взяти значення зі стеку, використовує _stack.pop()
    Побічний ефект: змінює значення _last_error
    :param variable: ім'я змінної
    :return: None
    """

def execute(self):

```

```

        """
        Метод виконує код програми, записаний у self._code.
        Повертає код останньої помилки або 0, якщо помилки
немає.

        Якщо є помилка, то показує її.
        Використовує словник функцій COMMAND_FUNCS
        :return: код останньої помилки або 0, якщо помилки немає
        """

def show_variables(self):
    """
    Метод показує значення усіх змінних пам'яті
    у форматі <змінна> = <значення>
    :return: None
    """

def get_value(self, variable):
    """
    Метод повертає значення змінної variable з пам'яті
    Якщо змінної у пам'яті немає або вона невизначена,
повертає None
    :param variable: ім'я змінної
    :return: float (or None)
    """

def add_to_storage(self, variable):
    """
    Метод додає зміну variable до пам'яті
    :param variable: ім'я змінної
    :return: None
    """

if __name__ == "__main__":
    code = [('LOADC', 1.0),
            ('SET', 'x'),
            ('LOADC', 1.0),
            ('SET', 'y'),
            ('LOADV', 'x'),
            ('LOADV', 'a'),
            ('MUL', None),
            ('SET', 't'),

```

```

        ('LOADC', 1.0),
        ('LOADV', 'x'),
        ('LOADV', 'y'),
        ('SUB', None),
        ('DIV', None),
        ('SET', 'z')]
interpreter = Interpreter(code, Storage())
interpreter.add_to_storage('x')
interpreter.add_to_storage('y')
interpreter.add_to_storage('a')
interpreter.add_to_storage('t')
interpreter.add_to_storage('z')
last_error = interpreter.execute()

success = last_error == 3

code = [ ('XXX', 1.0),
        ('SET', 'x'),
        ('LOADC', 1.0),
        ('SET', 'y')]
interpreter = Interpreter(code, Storage())
last_error = interpreter.execute()

success = success and last_error == 1

code = [ ('LOADC', 1.0),
        ('SET', 'x'),
        ('LOADC', 1.0),
        ('SET', 'y')]
interpreter = Interpreter(code, Storage())
last_error = interpreter.execute()

success = success and last_error == 2

code = [ ('LOADC', 2.0),
        ('SET', 'x'),
        ('LOADC', 1.0),
        ('SET', 'y'),
        ('LOADC', 1.0),
        ('LOADV', 'x'),
        ('LOADV', 'y'),
        ('SUB', None),
        ('DIV', None),
        ('SET', 'z')]
interpreter = Interpreter(code, Storage())
interpreter.add_to_storage('x')
interpreter.add_to_storage('y')

```



```

interpreter.add_to_storage('z')
last_error = interpreter.execute()

z = interpreter.get_value('z')
success = success and last_error == 0 and z == 1.0

print("Success =", success)

```

Тексти прикладів лінійних програм для інтерпретатора

Програма 1 – синтаксична помилка у 2 рядку

```

x = a + b

y = (x*x - 7*x) * (2 - 1.2 * x

```

Програма 2 – помилка при виконанні (ділення на 0)

```

x = 1
y = 2
u2 = x - 2*a
t = 2*x - y
z = (2*u2 + 3.2*x - 1.3*y)/t

```

Програма 3 – правильна програма

```

x = 1
y = 2
z = (x + y) * (x*x + 2*x*y + y*y)

```

## Групове завдання 18. Клавіатурний тренажер

### Текст завдання

Розробити програму – клавіатурний тренажер.

Програма має показувати користувачу по черзі слова, які випадковим чином вибираються з текстового файлу. У файлі кожне слово записано у окремому рядку. Користувач має ввести з клавіатури це слово за обмежений час.

Якщо слово введено правильно та у заданий термін, користувачу нараховується стільки балів, скільки літер є у слові.

За кожне неправильно введене слово користувачу нараховується штраф -1 бал.

Якщо слово введено правильно, але термін введення перевищено, бали не нараховуються.

Для реалізації програми описати клас `TypingTutor` та класи, що генерують виключення: `TypingError` – виключення при неправильно набраному слові, та `TimeError` – виключення при перевищенні ліміту часу.

Клас `TypingTutor` має містити конструктор та, як мінімум, 3 методи: `train`, `train_word`, `get_protocol`.

У конструкторі мають бути передані параметри: ім'я файлу зі словами, кількість слів для тренування, інтервал часу на введення слова у секундах.

Метод `train` має випадковим чином вибирати по черзі слова файлу та викликати метод `train_word`. Також має вести облік набраних балів. У цьому методі має бути обробка виключень `TypingError`, `TimeError`

Метод `train_word` має виводити слово та очікувати його введення з клавіатури. Далі аналізувати слово. Якщо є помилка введення – ініціювати виключення `TypingError`. Якщо перевищено ліміт часу – ініціювати виключення `TimeError`.

Метод `get_protocol` має повертати список кортежів, що містить для кожного слова з початку тренування такі дані: (<слово>, <правильно>, <час>, <бали>), де

<слово> - слово, яке мало бути введено,

<правильно> - чи правильно введене слово (булівське),

<час> - чи було дотримано ліміт часу при введенні слова,

<бали> - кількість набраних балів за слово (+ або -)

Основна частина програми має вводити (задавати) параметри, створювати об'єкт класу `TypingTutor`, виконувати тренування, показувати загальний результат та протокол.

Для вибору слова у випадковому порядку використати модуль `random`

Для вимірювання часу введення – функцію `time()` зі стандартного модуля `time`, яка повертає поточний час у вигляді дійсного числа, ціла частина якого – кількість секунд, що минули від початкової для Python дати.

Файл зі словами передано поштою. Кодування у файлі – `utf-8`

## Необхідні передумови

Засвоєння Теми 15 «Обробка помилок та виключних ситуацій».

## Мета завдання

Навчитись писати програми з класами обробки помилок, обробляти помилки у Python.

## Обмеження на виконання завдання

Немає.

## Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо описати класи обробки помилок, основну програму та клас `TypingTutor`.

### Зауваження щодо вирішення завдання

Перед заняттям треба розіслати студентам файл зі словами. Кожне слово має бути у окремому рядку файлу. Такий файл нескладно отримати з будь-якої статті українською мовою у мережі, завантаживши цю статтю та зберігши її у текстовий файл. Програма, що конвертує текстовий файл у файл зі словами, описана нижче.

### Рекомендації щодо перевірки результатів

Перевірити правильність нарахування балів при правильному введенні, помилковому введенні та перевищенні часу.

### Текст програми з розв'язками

Програма, що конвертує текстовий файл у файл зі словами. Вважається, що текст – у файлі `article_words.txt`.

```
import re

def prepare_string(string):
    # видалити всі символи-розділювачі
    string = re.sub(r' '[!?.,+;"]()\-A-Za-z0123456789]+'', '',
string)
    string = string.lower() # перевести до нижнього регістру
    # print(string)
    return string

file_name = "article_words.txt"
output_file = "words.txt"

with open(file_name, 'r', encoding='utf-8') as f:
    cnt = f.read()

cnt = prepare_string(cnt)

words_set = set(cnt.split())
words_list = [w for w in words_set if len(w) > 3]

print(words_list)

words = '\n'.join(words_list) + '\n'

with open(output_file, 'w', encoding='utf-8') as f:
    f.write(words)
```

Програма для клавіатурного тренажера

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```

import random
from time import time

class TypingError(Exception):
    pass

class TimeError(Exception):
    pass

class TypingTutor:

    def __init__(self, words_file, words_num, interval):
        self._words_num = words_num
        self._interval = interval
        self._protocol = list()
        with open(words_file, 'r', encoding='utf-8') as f:
            self._all_words = f.read().split()
            random.shuffle(self._all_words)
            self._words = self._all_words[:self._words_num]

    def train(self):
        for word in self._words:
            entered_ok = time_ok = True
            try:
                self.train_word(word)
                points = len(word)
            except TypingError:
                points = -1
                entered_ok = False
            except TimeError:
                points = 0
                time_ok = False

            self._protocol.append((word, entered_ok, time_ok,
points))

    def train_word(self, word):
        start_time = time()
        input_word = input(f"слово - {word}: ")
        end_time = time()
        if word != input_word:
            raise TypingError
        if end_time - start_time > self._interval:
            raise TimeError

```

```

def get_protocol(self):
    return self._protocol

def clear(self):
    self._protocol.clear()
    random.shuffle(self._all_words)
    self._words = self._all_words[:self._words_num]

if __name__ == '__main__':
    tutor = TypingTutor('words.txt', words_num=5, interval=4)
    tutor.train()
    protocol = tutor.get_protocol()
    print("Сума:", sum(list(zip(*protocol))[3]))
    print('Протокол')
    print(*protocol, sep='\n')

```

## Групове завдання 19. Складання кросвордів. Крок 1

### Текст завдання

У текстовому файлі зберігається сітка кросворду.

Сітка зберігається у текстовому файлі у вигляді, наприклад:

```

_v_____
h*****_
_ * _v_
_ * _ * _

```

Літерами v, h, b позначається початок слова по

h - горизонталі

v - вертикалі

b - горизонталі та вертикалі

Слова позначаються зірочками, порожні місця - підкресленнями

Усі рядки файлу мають бути однакової довжини

Треба описати модуль grid для обробки сітки. Цей модуль містить класи

OneCrossedWord – вже описано

WrongGrid – вже описано

Grid – треба описати

Коментарі щодо вмісту методів класу Grid – надано у модулі. Також у методі read за допомогою твердження про програму треба перевірити, що усі рядки у файлі мали однакову довжину.

Також потрібно написати програму, яка з використанням даних класів перевіряє

правильність заданої у файлі grid.txt сітки для кросворду зі словами space, ace, spot

та неправильність сітки для кросворду зі словами space, ace, step

### Необхідні передумови

Засвоєння Теми 16 «Ітератори та генератори».

### Мета завдання

Навчитись писати програми за специфікацією, що використовують ітератори та генератори у Python.

### Обмеження на виконання завдання

Немає.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо описати клас Grid та основну програму.

### Зауваження щодо вирішення завдання

Ще один приклад розробки програм за специфікацією. Це перше з 2 послідовних завдань, частина коду яких вже написана викладачем.

Перед заняттям треба розіслати студентам сітку кросворду та кістяк модуля grid. Кістяк модуля наведений після опису програми.

Щоб перевірити правильність твердження у завданні для 2 наборів слів, треба прочитати сітку кросворду з файлу, утворити множину кандидатів з усіх слів набору та послідовно викликати метод next\_match класу OneCrossedWord, доки множина не стане порожньою. Якщо при викликах завжди буде повернуто True, то сітка підходить для цього набору слів. Якщо хоча б 1 раз буде повернуто False, - сітка не підходить. Ми можемо так робити тому, що всі слова у наших наборах – різної довжини і двозначностей бути не може.

### Рекомендації щодо перевірки результатів

Перевірити правильність відповідей для наведених у завданні наборів слів.

### Текст програми з розв'язками

Модуль grid.

```
"""  
Модуль grid призначено для обробки сітки кросворду  
Модуль містить класи:  
OneCrossedWord  
Grid  
"""
```

```
from copy import copy
```

```

from collections import namedtuple

Cross = namedtuple(      # перетин слова з іншим словом
    "Cross",
    ["crossed_word",     # слово, що перетинається з даним,
                                     # об'єкт класу OneCrossedWord
    "crossed_index"])    # індекс перетину у слові, що
                        перетинається з даним

class OneCrossedWord:
    """
    Клас для обробки одного слова кросворду.
    """
    def __init__(self, no, pos, word_len, is_vert):
        self.no = no          # порядковий номер слова у
        кросворді: 1, 2 ...
        self.pos = pos        # позиція слова у сітці (рядок,
        стовпчик)
        self.len = word_len   # довжина слова
        self.is_vert = is_vert # чи розташовано слово по
        вертикалі
        self.crosses = dict()  # словник перетинів, ключ -
        індекс перетину
        self.word = ""        # слово
        self.candidates = set() # слова-кандидати на входження у
        кросворд

    def add_cross(self, index, cross):
        """
        Додати перетин для слова.
        :param index: індекс перетину у даному слові
        :param cross: іменований кортеж для перетину Cross
        :return: None
        """
        self.crosses[index] = cross

    def set_candidates(self, candidates):
        """
        Встановити множину слів-кандидатів
        :param candidates: множина слів-кандидатів
        :return: None
        """
        self.candidates = candidates

    def next_match(self):
        """

```

```

        Перевіряє допустимість входження у кросворд слів-
кандидатів
        Видаляє перевірені слова з множини кандидатів
        В разі успішності встановлює значення поля word
        :return: успішність bool
        """
        for candidate in copy(self.candidates):
            self.candidates.discard(candidate)
            if len(candidate) != self.len:
                continue

            if self._match(candidate):
                self.word = candidate
                return True

        return False

    def _match(self, candidate):
        """
        Перевіряє допустимість входження у кросворд одного
слова-кандидата
        :param candidate: кандидат str
        :return: успішність bool
        """
        for index, cross in self.crosses.items():
            letter = cross.crossed_word.get_crossed_letter(
                cross.crossed_index)
            if letter and letter != candidate[index]:
                return False

        return True

    def get_crossed_letter(self, index):
        """
        Повертає літеру, що стоїть (можливо) на перетині за
заданим індексом
        :param index: індекс літери
        :return: літера або None
        """
        if not self.word:
            return None

        return self.word[index]

    def clear(self):
        """
        Очищує слово

```



```

        :return:
        """
        self.word = ""

    def __str__(self):
        return "OneCrossedWord(no={}, pos={}, len={}, word={},
crosses={})" \
            .format(self.no, self.pos, self.len, self.word,
self.crosses)

class WrongGrid(Exception):
    """
    Клас виключення для неправильної сітки
    """
    pass

class Grid:
    """
    Клас обробки сітки кросворду
    Сітка зберігається у текстовому файлі у вигляді
    v
    -----
    h*****
    -----
    *      v
    -----
    *      *
    -----
    Літерами v, h, b позначається початок слова по
    h - горизонталі
    v - вертикалі
    b - горизонталі та вертикалі
    Слова позначаються зірочками порожні місця - підкресленнями
    Усі рядки файлу мають бути однакової довжини
    Після завантаження з файлу сітка міститься у двовимірному
    масиві символів
    """
    def __init__(self, filename):
        self._filename = filename    # ім'я файлу сітки
        self._grid = []              # масив сітки
        self._words = list()         # список слів - об'єктів
        класу OneCrossedWord
        self._cur_index = 0          # індекс поточного слова

    def read(self):
        """
        Читає сітку з файлу у масив
        Будує список слів, для кожного слова будує словник
        перетинів

```

```

        У масиві замінює символи початку слова на номер слова
        :return: None
        """
        with open(self._filename, 'r') as f:
            for line in f:
                self._grid.append(list(line.strip()))

        assert all(len(x) == len(self._grid[0]) for x in
self._grid), \
            "Invalid length of line in grid"

        self._make_words()
        self._make_crosses()

    def _make_words(self):
        n = len(self._grid)
        m = len(self._grid[0])
        no = 0
        for i in range(n):
            for j in range(m):
                if self._grid[i][j] not in ('h', 'v', 'b'):
                    continue

                no += 1
                pos = (i, j)
                if self._grid[i][j] in ('h', 'b'):
                    is_vert = False
                    w_str = ''.join(self._grid[i][j:])
                    w_end = w_str.find('_')
                    word_len = w_end if w_end >= 0 else
len(w_str)

                    self._words.append(
                        OneCrossedWord(no, pos, word_len,
is_vert))

                if self._grid[i][j] in ('v', 'b'):
                    is_vert = True
                    w_str = ''.join([self._grid[k][j] for k in
range(i, n)])

                    w_end = w_str.find('_')
                    word_len = w_end if w_end >= 0 else
len(w_str)

                    self._words.append(
                        OneCrossedWord(no, pos, word_len,
is_vert))

                self._grid[i][j] = str(no)

    def _make_crosses(self):

```

```

    for word in self._words:
        if not word.is_vert:
            self._make_crosses_for_word(word)

def _make_crosses_for_word(self, word):
    n = len(self._grid)
    m = len(self._grid[0])
    i, j = word.pos
    for l in range(j, j + word.len):
        k = i
        while k > 0 and self._grid[k-1][l] != '_':
            k -= 1
        if k < i or i < n - 1 and self._grid[i+1][l] != '_':
            other = self.find_word_by_pos((k, l))
            if not other:
                raise Exception("Something wrong")
            index = l - j
            other_index = i - k
            word.add_cross(index, Cross(other, other_index))
            other.add_cross(other_index, Cross(word, index))

def find_word_by_pos(self, pos, is_vert=True):
    """
    Знаходить слово за позицією у масиві
    :param pos: кортеж (рядок, стовпчик)
    :param is_vert: чи розташоване слово по вертикалі
    :return: слово (str) або None
    """
    for word in self._words:
        if word.pos == pos and word.is_vert == is_vert:
            return word

    return None

def __iter__(self):
    """
    Метод ітератора
    :return: повертає себе
    """
    return self

def __next__(self):
    """
    Метод ітератора
    повертає наступне слово
    :return:
    """

```

```

    if self._cur_index >= len(self._words):
        raise StopIteration

    word = self._words[self._cur_index]
    self._cur_index += 1
    return word

def undo_next(self):
    """
    Відмінити останнє передане слово та повернутись до
попереднього,
змінює поточний індекс слова
Ініціює помилку WrongGrid, якщо не можемо повернутись
до попереднього слова
:return:
    """
    if self._cur_index <= 0:
        raise WrongGrid("Wrong grid")

    self._cur_index -= 1

def reset(self):
    """
    Перезапустити ітератор спочатку
:return: None
    """
    self._cur_index = 0

def show(self):
    """
    Показати сітку, замінивши символи початку слова на номер
слова,
та '_' на ' '
:return: None
    """
    for row in self._grid:
        print(''.join(row).replace('_', ' '))

def show_words(self):
    """
    Показати слова
:return: None
    """
    for word in self._words:
        print(word)

```

```

if __name__ == "__main__":
    g = Grid("grid.txt")
    g.read()
    g.show()
    g.show_words()

```

Сітка кросворду у файлі grid.txt

```

      v
h*****
      *      v
      *      *
h*****
      *      *
      *      *
      h***
      *

```

Кістяк модуля grid

```

"""
Модуль grid призначено для обробки сітки кросворду
Модуль містить класи:
OneCrossedWord
Grid
"""

from copy import copy
from collections import namedtuple

Cross = namedtuple(      # перетин слова з іншим словом
    "Cross",
    ["crossed_word",      # слово, що перетинається з даним,
    # об'єкт класу OneCrossedWord
    "crossed_index"])    # індекс перетину у слові, що
    перетинається з даним

class OneCrossedWord:
    """
    Клас для обробки одного слова кросворду.
    """
    def __init__(self, no, pos, word_len, is_vert):
        self.no = no      # порядковий номер слова у
        кросворді: 1, 2 ...
        self.pos = pos    # позиція слова у сітці (рядок,
        стовпчик)

```

```

        self.len = word_len      # довжина слова
        self.is_vert = is_vert   # чи розташовано слово по
вертикалі
        self.crosses = dict()     # словник перетинів, ключ -
індекс перетину
        self.word = ""           # слово
        self.candidates = set()   # слова-кандидати на входження у
кросворд

    def add_cross(self, index, cross):
        """
        Додати перетин для слова.
        :param index: індекс перетину у даному слові
        :param cross: іменований кортеж для перетину Cross
        :return: None
        """
        self.crosses[index] = cross

    def set_candidates(self, candidates):
        """
        Встановити множину слів-кандидатів
        :param candidates: множина слів-кандидатів
        :return: None
        """
        self.candidates = candidates

    def next_match(self):
        """
        Перевіряє допустимість входження у кросворд слів-
кандидатів
        Видаляє перевірені слова з множини кандидатів
        В разі успішності встановлює значення поля word
        :return: успішність bool
        """
        for candidate in copy(self.candidates):
            self.candidates.discard(candidate)
            if len(candidate) != self.len:
                continue

            if self._match(candidate):
                self.word = candidate
                return True

        return False

    def _match(self, candidate):
        """

```

```

        Перевіряє допустимість входження у кросворд одного слова-кандидата
        :param candidate: конадидат str
        :return: успішність bool
        """
        for index, cross in self.crosses.items():
            letter = cross.crossed_word.get_crossed_letter(
                cross.crossed_index)
            if letter and letter != candidate[index]:
                return False

        return True

    def get_crossed_letter(self, index):
        """
        Повертає літеру, що стоїть (можливо) на перетині за заданим індексом
        :param index: індекс літери
        :return: літера або None
        """
        if not self.word:
            return None

        return self.word[index]

    def clear(self):
        """
        Очищує слово
        :return:
        """
        self.word = ""

    def __str__(self):
        return "OneCrossedWord(no={}, pos={}, len={}, word={}, crosses={})" \
            .format(self.no, self.pos, self.len, self.word, self.crosses)

class WrongGrid(Exception):
    """
    Клас виключення для неправильної сітки
    """
    pass

class Grid:

```

```

"""
Клас обробки сітки кросворду
Сітка зберігається у текстовому файлі у вигляді
  v
  h*****
  *      v
  *      *
Літерами v, h, b позначається початок слова по
    h - горизонталі
    v - вертикалі
    b - горизонталі та вертикалі
Слова позначаються зірочками порожні місця - підкресленнями
Усі рядки файлу мають бути однакової довжини
Після завантаження з файлу сітка міститься у двовимірному
масиві символів
"""

def __init__(self, filename):
    self._filename = filename      # ім'я файлу сітки
    self._grid = []               # масив сітки
    self._words = list()          # список слів - об'єктів
класу OneCrossedWord
    self._cur_index = 0           # індекс поточного слова

def read(self):
    """
    Читає сітку з файлу у масив
    Будує список слів, для кожного слова будує словник
перетинів
    У масиві замінює символи початку слова на номер слова
    :return: None
    """

def find_word_by_pos(self, pos, is_vert=True):
    """
    Знаходить слово за позицією у масиві
    :param pos: кортеж (рядок, стовпчик)
    :param is_vert: чи розташоване слово по вертикалі
    :return: слово (str) або None
    """

def __iter__(self):
    """
    Метод ітератора
    :return: повертає себе
    """

```



```

def __next__(self):
    """
    Метод ітератора
    повертає наступне слово
    :return:
    """

def undo_next(self):
    """
    Відмінити останнє передане слово та повернутись до
попереднього,
    змінює поточний індекс слова
    Ініціює помилку WrongGrid, якщо не можемо повернутись
до попереднього слова
    :return:
    """

def reset(self):
    """
    Перезапустити ітератор спочатку
    :return: None
    """

def show(self):
    """
    Показати сітку, замінивши символи початку слова на номер
слова,
    та '_' на ' '
    :return: None
    """

def show_words(self):
    """
    Показати слова
    :return: None
    """

if __name__ == "__main__":
    g = Grid("grid.txt")
    g.read()
    g.show()
    g.show_words()

```

## Групове завдання 20. Складання кросвордів. Крок 2

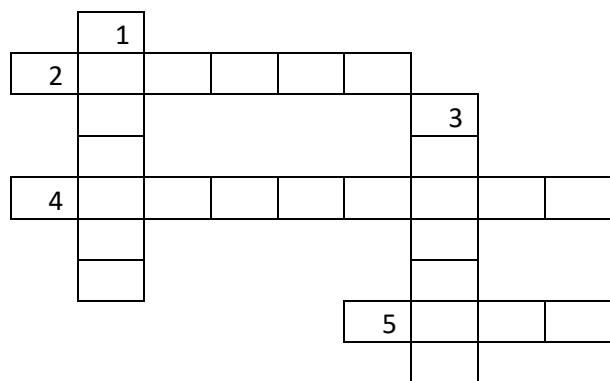
### Текст завдання

У текстовому файлі збережено слова та їх означення (описи). У одному рядку файлу – слово у лапках, двокрапка та означення слова у лапках.

Потрібно скласти програму, яка за заданим файлом та заданою сіткою складає та показує кросворд, що містить слова зі списку.

Використати класи з завдання 19 (для довідки завдання 19 дублюється нижче). Класи з завдання 19 у доданому архіві описано повністю, але можна використати свої описані класи, якщо вони працюють правильно.

Наприклад



Побудувати класи Crossword та Thesaurus (заповнити кодом незаповнені методи). Можна додавати свої внутрішні методи.

Показувати сітку та слова можна у текстовому режимі (тоді сітка може бути показана зірочками \*, а номери слів - цифрами) або за допомогою turtle.

Треба описати необхідні класи, створити кросворд для зображеної сітки (файл grid.txt) зі слів з файлу thesaurus.txt.

### Необхідні передумови

Засвоєння Теми 16 «Ітератори та генератори».

### Мета завдання

Навчитись писати програми за специфікацією, що використовують ітератори та генератори у Python.

### Обмеження на виконання завдання

Немає.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо описати класи Crossword та Thesaurus.

### Зауваження щодо вирішення завдання

Ще один приклад розробки програм за специфікацією. Це друге з 2 послідовних завдань, частина коду яких вже написана викладачем.

Перед заняттям треба розіслати студентам сітку кросворду (відрізняється від попереднього завдання), кістяк модулів crossword та thesaurus а також сам тезаурус у текстовому файлі. Кістяк модулів наведений після опису програми.

В якості тезаурусу можна взяти вільно розповсюджуваний тезаурус у мережі або створити свій. Формат файлу тезаурусу має відповідати специфікації.

### Рекомендації щодо перевірки результатів

Якщо класи Thesaurus та Crossword будуть реалізовані правильно, то після запуску головної програми з модуля crossword ми побачимо сітку кросворду, список слів – об'єктів OneCrossedWord а також описи слів кросворду по горизонталі та вертикалі.

### Текст програми з розв'язками

Модуль crossword.

```
from thesaurus import Thesaurus
from grid import Grid

class CrossWord:
    """
    Будує кросворд за заданою сіткою та тезаурусом, якщо це
    можливо
    """
    def __init__(self, thes_filename, grid_filename):
        self._thesaurus = Thesaurus(thes_filename) # тезаурус
        self._grid = Grid(grid_filename)           # сітка

        self._thesaurus.read()
        self._grid.read()
        self._descriptions_vert = list()           # описи слів
        по вертикалі
        self._descriptions_hor = list()             # описи лів
        по горизонталі

    def check_cross_word(self):
        """
        Перевіряє можливість побудови та будує кросворд
        Будує описи слів
        :return:
        """
        count = 0
        words_in_use = []
        for word in self._grid:
```

```

        if word.no > count:
            count += 1
            words_in_use.append("")
            candidates =
self._thesaurus.get_words_with_len(word.len) - \
                                set(words_in_use)
            word.set_candidates(candidates)
        if not word.next_match():
            word.clear()
            words_in_use.pop()
            count -= 1
            self._grid.undo_next()
            self._grid.undo_next()
        else:
            words_in_use[-1] = word.word

    self._grid.reset()
    for word in self._grid:
        if word.is_vert:
            self._descriptions_vert.append(
                (word.no,
self._thesaurus.get_description(word.word)))
        else:
            self._descriptions_hor.append(
                (word.no,
self._thesaurus.get_description(word.word)))

    def show(self):
        """
        Показує сітку кросворду та описи слів
        :return:
        """
        self._grid.show()
        self._grid.show_words()
        print("Horizontal", *self._descriptions_hor, sep='\n')
        print("Vertical", *self._descriptions_vert, sep='\n')

if __name__ == "__main__":
    cw = CrossWord("thesaurus.txt", "grid.txt")
    cw.check_cross_word()
    cw.show()

```

Модуль thesaurus.

```
class Thesaurus:
```

```

"""
Клас працює з тезаурусом (тлумачним словником)
Словник записано у файлі у форматі:
"слово": "опис"
Кожне слово разом з описом записано у окремому рядку файлу
"""

def __init__(self, filename):
    self._filename = filename      # ім'я файлу тезаурусу
    self._thesaurus = dict()      # тезаурус - словник з
    # та значеннями - описами
    # ключами - словами

    self._words_lists = dict()    # словник списків слів
    # заданої довжини
    # ключі - довжини слів,
    # значення - списки слів

def read(self):
    """
    Читає тезаурус з файлу, будує словник self._thesaurus та
    списки слів
    :return:
    """
    with open(self._filename, 'r') as f:
        for line in f:
            line = line.strip()
            # print(line)
            if not line:
                break

            word, description = line.split(':')
            self._thesaurus[word.strip('")] =
description.strip('')

            self._build_words_lists()

def _build_words_lists(self):
    maxlen = len(max(self._thesaurus, key=len))
    for word_len in range(1, maxlen + 1):
        self._words_lists[word_len] = [w for w in
self._thesaurus
                                if len(w) ==
word_len]

def get_words_with_len(self, word_len):
    """
    Повертає список слів заданої довжини word_len

```

```

        :param word_len: довжина слова
        :return: список слів [list]
        """
        return set(self._words_lists.get(word_len, []))

    def get_description(self, word):
        """
        Повертає опис слова word
        :param word: слово
        :return: опис [str]
        """
        return self._thesaurus.get(word, "")

if __name__ == "__main__":
    t = Thesaurus("thesaurus.txt")
    t.read()
    for i in range(1, 6):
        words_i = t.get_words_with_len(i)
        for word in words_i:
            print(word, t.get_description(word))

```

Сітка кросворду grid.txt

```

v
h*****
*       v
*       *
h*****
*       *
*       *
      h***
      *

```

Кістяк модуля crossword з реалізованим конструктором класу CrossWord та головною частиною програми.

```

from thesaurus import Thesaurus
from grid import Grid

```

```

class CrossWord:
    """
    Буде кросворд за заданою сіткою та тезаурусом, якщо це
    МОЖЛИВО
    """

```

```

def __init__(self, thes_filename, grid_filename):
    self._thesaurus = Thesaurus(thes_filename) # тезаурус
    self._grid = Grid(grid_filename)          # сітка

    self._thesaurus.read()
    self._grid.read()
    self._descriptions_vert = list()          # описи слів
по вертикалі
    self._descriptions_hor = list()           # описи лів
по горизонталі

def check_cross_word(self):
    """
    Перевіряє можливість побудови та будує кросворд
    Будує описи слів
    :return:
    """

def show(self):
    """
    Показує сітку кросворду та описи слів
    :return:
    """

if __name__ == "__main__":
    cw = CrossWord("thesaurus.txt", "grid.txt")
    cw.check_cross_word()
    cw.show()

```

Кістяк модуля thesaurus з реалізованим конструктором класу Thesaurus

```

class Thesaurus:
    """
    Клас працює з тезаурусом (тлумачним словником)
    Словник записано у файлі у форматі:
    "слово": "опис"
    Кожне слово разом з описом записано у окремому рядку файлу
    """

    def __init__(self, filename):
        self._filename = filename # ім'я файлу тезаурусу
        self._thesaurus = dict()  # тезаурус - словник з
ключами - словами                # та значеннями - описами
слів

```

```

        self._words_lists = dict() # СЛОВНИК СПИСКІВ СЛІВ
заданої довжини
                                     # ключі - довжини слів,
значення - СПИСКИ СЛІВ

    def read(self):
        """
        Читає тезаурус з файлу, будує словник self._thesaurus та
СПИСКИ СЛІВ
        :return:
        """

    def get_words_with_len(self, word_len):
        """
        Повертає список слів заданої довжини word_len
        :param word_len: довжина слова
        :return: список слів [list]
        """

    def get_description(self, word):
        """
        Повертає опис слова word
        :param word: слово
        :return: опис [str]
        """

if __name__ == "__main__":
    t = Thesaurus("thesaurus.txt")
    t.read()
    for i in range(1, 6):
        words_i = t.get_words_with_len(i)
        for word in words_i:
            print(word, t.get_description(word))

```

## Групове завдання 21. Розрахунок матеріалів для будинку

### Текст завдання

Будується будинок прямокутної форми з двоскатним дахом. Матеріал фундаменту – бетон. Матеріал стін – газоблок. Матеріал даху металочерепиця.

Описати класи: Будинок, Стіна, Фундамент, Дах, Вікно, Двері. Описати також клас-ітератор, який повертає по черзі усі стіни будинку. За потребою, можна використовувати й інші класи.



Використати твердження про програми assert для перевірки, що усі стіни мають однакову висоту, а протилежні стіни – однакову ширину.

Усі класи мають містити як мінімум конструктор та метод show – показати.

Показ будинку здійснити з використанням turtle відразу з 4 боків (по окремих стінах).

Дані про розміри будинку а також фундамент, дах, стіни, вікна, двері зберігаються у текстовому файлі у такому вигляді:

У першому рядку – загальні дані про фундамент

У другому рядку – загальні дані про дах.

У наступному рядку – дані фронтальної стіни ("front", ...)

У наступних рядках – дані про двері ("door", ...) та вікна ("window", ...), які виходять на цю стіну.

Далі – аналогічно дані про інші стіни будинку ("rear", "left", "right")

Для кожної стіни вказують її габаритні розміри. Бокові стіни мають трикутні фронтони під дахом. Розміри вказують без урахування фронтонів.

Для кожного вікна та двері задаються положення лівого нижнього кута відносно лівого нижнього кута стіни та габаритні розміри.

Для даху вказують вихід за стіну з кожного з 4 боків та кут нахилу даху у градусах.

Для фундаменту (стрічкового) вказують його товщину, глибину відносно землі та висоту над землею.

Розмір газоблоку: 0.6 x 0.4 x 0.2 м, товщина стін – 0.4 м

Робоча ширина листа металочерепиці 1.05 м

Треба прочитати параметри будинку з файлу, показати будинок а також розрахувати:

- Об'єму бетону
- Кількість газоблоків (з урахуванням допуску додатково 10% від мінімально необхідної кількості)
- Кількість та довжини листів черепиці (черепиця відрізається потрібного розміру по довжині)

### Необхідні передумови

Засвоєння Теми 16 «Ітератори та генератори».

### Мета завдання

Навчитись писати програми, що використовують ітератори та генератори, обробку помилок у Python.

### Обмеження на виконання завдання

Немає.

Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо описати різні класи з завдання.

Зауваження щодо вирішення завдання

Повернення до теми першого завдання вже на новій мовній базі, тобто, з використанням класів та зі збільшенням можливостей.

Рекомендації щодо перевірки результатів

Задати прості дані та перевірити правильність відповідей.

## Групове завдання 22. Ланцюговий код. Крок 1

Текст завдання

Бінарне зображення на прямокутнику  $[(0,0), (m-1, n-1)]$  складається з точок. Кожна точка може мати значення 1 (зафарбовано) або 0 (не зафарбовано).

Для стиснення бінарного зображення використовують ланцюговий код, який визначається наступним чином: для послідовності зафарбованих точок у рядку  $i$ , починаючи з позиції  $j$ , довжиною  $k$  зберігають кортеж  $(i, j, k)$ .

Нехай у текстовому файлі один рядок відповідає одному рядку зображення. Зафарбовані точки позначені зірочками ('\*'), не зафарбовані - крапками ('.'). Наприклад, початковий рядок файлу може мати вигляд:

```
.... *** ... ***** ..... **** .....
```

Для рядка файлу, зображеного вище, маємо таку послідовність ланцюгових кодів: (0, 4, 3), (0, 10, 8), (0, 23, 4)

Описати клас ChainCodeReader, який читає бінарне зображення з текстового файлу (ім'я файлу передається у конструкторі) та формує послідовність ланцюгових кодів. Цей клас також має бути ітератором то повертати усі прочитані коди у порядку слідування. Окрім цього клас має містити властивість (property) codes, що повертає список прочитаних кодів

Для показу за допомогою turtle одну зафарбовану точку показувати зафарбованим квадратом розміром  $a$  (для незафарбованої точки відповідно пропустити квадрат розміром  $a$ ).

Описати клас ChainCodePicture, який має метод show для показу бінарного зображення, представленого у вигляді послідовності ланцюгових кодів, за допомогою turtle. У конструктор класу передаються параметри:

- codes - послідовність кодів
- scale – масштаб відображення точки (значення  $a$ )
- color – колір відображення

Врахувати, що у turtle вісь OY спрямована угору.

Для зображення одного зафарбованого квадрату довжиною  $a$  чорним кольором, починаючи з поточної позиції, можна використати такі команди turtle:

```
turtle.color('black', 'black')
```

```
turtle.begin_fill()
```

```
for i in range(4):
```

```
    turtle.fd(a)
```

```
    turtle.right(90)
```

```
turtle.end_fill()
```

З використанням класів ChainCodeReader та ChainCodePicture розв'язати задачу. Нехай у текстовому файлі записано бінарне зображення для сьогоднішньої дати (у форматі “dd mm yyyy”, dd - день, mm – місяць, yyyy - рік). Перетворити це зображення у ланцюговий код та показати.

### Необхідні передумови

Засвоєння Тем 16 «Ітератори та генератори», 17 «Декоратори».

### Мета завдання

Навчитись писати програми, що використовують ітератори та генератори, стандартні декоратори (property) у Python.

### Обмеження на виконання завдання

Немає.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо описати різні класи з завдання.

### Зауваження щодо вирішення завдання

Це завдання також складається з 2 кроків (2 послідовних занять). Кінцева мета – побудувати класи для обробки ланцюгових кодів та визначення фігур у бінарному зображенні.

Між першим та другим заняттям треба надати можливість студентам обмінюватись написаними програмами, щоб до другого заняття всі мали працюючі класи ChainCodeReader та ChainCodePicture.

### Рекомендації щодо перевірки результатів

Наочно перевірити правильність зображення а також впевнитись у реалізації описаних класів та методів.

### Текст програми з розв'язками

Модуль chain\_code

```
import turtle
```

```
X_START = -200
```

```
Y_START = 200
```

```
class ChainCodeReader:
    """
    Клас читає бінарне зображення, яке записано у текстовому
    файлі
    зірочками та крапками, та повертає послідовність ланцюгових
    кодів
    """
    def __init__(self, filename):
        self._codes = list()
        with open(filename, 'r') as f:
            for i, line in enumerate(f):
                self._extract_codes(line.strip(), i)
        self._index = 0

    def _extract_codes(self, line, row):
        """
        Виділити коди з рядка файлу.
        Модифікує поле self._codes
        :param line: рядок файлу без '\n' [str]
        :param row: номер рядка зображення [int]
        :return: None
        """
        line += '.'
        in_chain = False
        for i, c in enumerate(line):
            if c == '*':
                if not in_chain:
                    col = i
                    length = 0
                    in_chain = True
                    length += 1
            else:
                if in_chain:
                    self._codes.append((row, col, length))
                    in_chain = False

    @property
    def codes(self):
        """
        Властивість повертає список ланцюгових кодів зображення
        :return: список кодів [list]
        """
        return self._codes
```

```

def __iter__(self):
    """
    Метод для підтримки ітераційного протоколу
    Повертає себе в якості ітератора
    :return: self [ChainCodeReader]
    """
    return self

def __next__(self):
    """
    Метод для підтримки ітераційного протоколу
    Повертає наступний ланцюговий код
    :return:
    """
    if self._index >= len(self._codes):
        raise StopIteration

    elem = self._codes[self._index]
    self._index += 1
    return elem

def reset(self):
    """
    Реініціалізувати ітератор
    :return: None
    """
    self._index = 0


class ChainCodePicture:
    """
    Клас зображує бінарне зображення у ланцюгових кодах
    за допомогою turtle
    """
    def __init__(self, codes, scale, color):
        self._codes = codes          # ланцюгові коди
        self._scale = scale          # масштаб зображення
        (кількість пікселів
        self._color = color           # у 1 точці
        self._x_start = X_START      # колір зображення
        self._y_start = Y_START      # зсув зображення по x
        # зсув зображення по y

    def show(self):
        """
        Показати зображення
        :return: None

```

```

        """
        turtle.up()
        turtle.home()
        turtle.delay(0)
        turtle.color(self._color, self._color)
        for code in self._codes:
            self._show_code(code)

    def _show_point(self, row, col):
        """
        Показати 1 точку зображення у вигляді квадрата
        :param row: номер рядка
        :param col: номер стовпчика
        :return: None
        """
        turtle.up()
        turtle.setpos(self._x_start + col * self._scale,
                      self._y_start - row * self._scale)
        turtle.down()
        turtle.begin_fill()
        for i in range(4):
            turtle.fd(self._scale)
            turtle.right(90)
        turtle.end_fill()

    def _show_code(self, code):
        """
        Показати 1 ланцюговий код
        :param code:
        :return:
        """
        row, start_col, length = code
        for i in range(start_col, start_col + length):
            self._show_point(row, i)

if __name__ == '__main__':
    reader = ChainCodeReader('ht23.txt')
    picture = ChainCodePicture(reader.codes, 5, 'black')
    picture.show()

```

## Групове завдання 23. Ланцюговий код. Крок 2

### Текст завдання

Завдання продовжує завдання 22

Бінарне зображення на прямокутнику  $[(0,0), (m, n)]$  складається з точок. Кожна точка може мати значення 1 (зафарбовано) та 0 (не зафарбовано).

Для стиснення бінарного зображення використовують ланцюговий код, який визначається наступним чином: для послідовності зафарбованих точок у рядку  $i$ , починаючи з позиції  $j$ , довжиною  $k$  зберігають кортеж  $(i, j, k)$ .

Фігура у ланцюгових кодах визначається наступним чином: два сусідніх рядки фігури мають хоча б один перетин (тобто, мають ланцюгові коди, які перетинаються). Приклад перетину – на рисунку нижче

```
.....*****.....  
.....***.....
```

Описати клас ChainCodeFigure, який зберігає одну фігуру у ланцюгових кодах. Цей клас повинен мати методи:

- Конструктор – створює порожню фігуру
- add\_code – додати ланцюговий код code до фігури
- weight – обчислити та повернути «вагу» фігури, кількість точок у фігурі
- mass\_center – обчислити та повернути центр мас фігури (суми значень по відповідних координатах розділити на вагу)
- codes – властивість (property), повертає усі коди фігури
- intersects – перевіряє, чи перетинається фігура з заданим ланцюговим кодом code
- merge – злити фігуру з іншою фігурою figure
- \_\_lt\_\_ - чи менше поточна Фігура1, ніж Фігура2, other. Фігура1 < Фігура2, якщо вона знаходиться вище (спочатку) та лівіше (потім), ніж Фігура2
- show – показати фігуру у масштабі scale кольором color

З використанням класів ChainCodeReader, ChainCodePicture, ChainCodeFigure розв'язати задачу. У текстовому файлі в архіві ht23.zip записано бінарне зображення. Порахувати та показати (print) кількість фігур, координати центру мас та вагу кожної фігури. Показати зображення кожної окремої фігури у turtle у заданому масштабі чорним кольором.

### Необхідні передумови

Засвоєння Тем 16 «Ітератори та генератори», 17 «Декоратори».

### Мета завдання

Навчитись писати програми, що використовують ітератори та генератори, стандартні декоратори (property) у Python.

### Обмеження на виконання завдання

Немає.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо описати різні методи класу ChainCodeFigure з завдання.

### Зауваження щодо вирішення завдання

Окрім самого завдання перед початком заняття студентам треба розіслати текстовий файл з бінарним зображенням фігур, які треба виділити.

Найбільш складним у завданні є функція `build_figures` побудови списку фігур. Можливо, тут треба підказати командам, яким чином можна це зробити та які структури даних краще використати.

### Рекомендації щодо перевірки результатів

Наочно перевірити правильність зображення а також впевнитись у реалізації описаних класів та методів.

### Текст програми з розв'язками

Модуль `chain_code_figure`

```
import time
from chain_code import ChainCodePicture, ChainCodeReader

SCALE = 10
COLOR = "black"

class ChainCodeFigure:
    def __init__(self, codes=()):
        self._codes = list(codes)
        self._weight = sum(c[2] for c in self._codes)

    def add_code(self, code):
        self._codes.append(code)
        self._weight += code[2]

    @property
    def codes(self):
        return self._codes

    def intersects(self, code):
        code_row, code_col, code_length = code
        for row, col, length in self._codes:
            if abs(row - code_row) == 1 and (
                code_col <= col < code_col + code_length or
                col <= code_col < col + length or
                code_col <= col + length - 1 < code_col +
code_length or
                col <= code_col + code_length - 1 < col +
length):
                return True

        return False

    def merge(self, other):
        self._codes.extend(other.codes)
```



```

        self._weight += other._weight
        self._codes.sort()

    def weight(self):
        return self._weight

    def rect(self):
        x_min = min(c[1] for c in self._codes)
        y_min = min(c[0] for c in self._codes)
        x_max = max(c[1] + c[2] - 1 for c in self._codes)
        y_max = max(c[0] for c in self._codes)
        return x_min, y_min, x_max, y_max

    def start_row(self):
        return min(c[0] for c in self._codes)

    def start_col(self):
        return min(c[1] for c in self._codes)

    def mass_center(self):
        weight = self.weight()
        assert weight, "Can't calculate mass center for empty
figure"

        x_sum = y_sum = 0
        for row, col, length in self._codes:
            y_sum += row * length
            x_sum += sum(range(col, col + length))
        return x_sum / weight, y_sum / weight

    def __lt__(self, other):
        return self.start_row() < other.start_row() or \
            (self.start_row() == other.start_row() and
             self.start_col() < other.start_col())

    def show(self):
        cp = ChainCodePicture(self._codes, SCALE, COLOR)
        cp.show()

if __name__ == '__main__':
    def build_figures(reader):
        figures = list()
        for code in reader:
            intersected = list()
            for figure in figures:
                if figure.intersects(code):

```



```

..... * * * * ..... * * * * * ..... * * * * . * * * * .....
..... * * * * * * * * ..... * * * * ..... * * * * .....
..... * * * * ..... * ..... * * * * .....
..... * * * * ..... * * * * .....
..... * * * * .....
.....
.....
.....

```

## Групове завдання 24. Фільтрація та відновлення рядків

### Текст завдання

Є функція, що повертає список слів.

Побудувати «фільтри» у вигляді декораторів, які «псують» слова з цього списку наступним чином:

1. Додають до коду 1 символу слова деяке задане число.
2. Видаляють 1 символ слова
3. Вставляють 1 символ у слово

Отримати випадковий рядок з файлу “text.txt”, запам’ятати його в окремому текстовому файлі, перетворити його у список слів та пропустити через «фільтри». Кожне слово word може проходити через фільтри не більше  $\text{len}(\text{word}) // 3$  разів.

Утворити зі списку відфільтрованих слів рядок (слова мають розділятися пропусками) та передати іншій команді, вказавши окрім рядка назву своєї команди (A, B, C, D, E) та порядковий номер фільтрованого повідомлення.

Інша команда має спробувати відтворити все або максимальну частину первинного повідомлення.

Відтворення треба робити за допомогою алгоритму, що обчислює відстань Левенштейна між 2 рядками. Для цього з файлу “text.txt” отримати усі різні слова та знайти для кожного слова повідомлення слово з файлу з мінімальною відстанню Левенштейна.

Перемагає команда, яка відтворить максимальну кількість повідомлень інших команд.

Для розрахунку відстані Левенштейна найчастіше застосовують простий алгоритм, в якому використовується матриця розміром  $(n + 1) * (m + 1)$ , де  $n$  і  $m$  - довжини порівнюваних рядків. Окрім цього вартість операцій вилучення, заміни та вставки вважається однаковою. Для конструювання матриці використовують таке рекурсивне рівняння:

$D_{0,0}=0$

|  $D_{i-1,j-1} + 0$  (equal)

|  $D_{i-1,j-1} + 1$  (replace)

$D_{i,j}=\min\{$

|  $D_{i-1,j} + 1$  (insert)

|  $D_{i,j-1} + 1$  (delete)

У [псевдокодi](#) алгоритм виглядає так:

```
int LevenshteinDistance(char str1[1..lenStr1], char str2[1..lenStr2])
    // d таблиця кількість рядків = lenStr1+1 та кількість стовпців = lenStr2+1
    declare int d[0..lenStr1, 0..lenStr2]
    // i та j використовуються для індексування позиції у str1 та у str2
    declare int i, j, cost

    for i from 0 to lenStr1
        d[i, 0] := i
    for j from 0 to lenStr2
        d[0, j] := j

    for i from 1 to lenStr1
        for j from 1 to lenStr2
            if str1[i] = str2[j] then cost := 0    //однакові
                else cost := 1    //заміна
            d[i, j] := minimum(
                d[i-1, j ] + 1,    // вилучення
                d[i , j-1] + 1,    // вставка
                d[i-1, j-1] + cost // заміна або однакові
            )

    return d[lenStr1, lenStr2] //значення відстані Левенштайна в останній клітинці матриці
```

### Необхідні передумови

Засвоєння Теми 17 «Декоратори».

### Мета завдання

Навчитись писати програми, що використовують декоратори у Python.

### Обмеження на виконання завдання

Немає.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо описати різні фільтри та відтворення рядка.

### Зауваження щодо вирішення завдання

Окрім самого завдання перед початком заняття студентам треба розіслати текстовий файл з повідомленнями.

Є сенс давати це завдання, якщо у групі студентів є декілька приблизно рівних команд. Також можна подумати над критерієм перемоги, оскільки команда залежить від спроможності інших виконати хоча б першу частину завдання.

### Рекомендації щодо перевірки результатів

Наочно перевірити правильність зображення а також впевнитись у реалізації описаних класів та методів.

### Текст програми з розв'язками

Рекурсивна функція обчислення відстані Левенштейна

*# відстань Левенштейна*

```
def levenstein(s1, s2):
    if not s1 and not s2:
        d = 0
    elif not s1:
        d = len(s2)
    elif not s2:
        d = len(s1)
    else:
        cost = 0 if s1[-1] == s2[-1] else 1
        d = min(levenstein(s1[:-1], s2[:-1]) + cost,
                levenstein(s1[:-1], s2) + 1,
                levenstein(s1, s2[:-1]) + 1)
    return d

s1 = input('s1= ')
s2 = input('s2= ')
d = levenstein(s1, s2)
print('d=', d)
```

## Групове завдання 25. Обмін повідомленнями між програмами

### Текст завдання

Скласти 2 програми. Програма1 вводить повідомлення, а Програма2 на основі цих повідомлені викликає методи заданого класу Recipient. Програма1 та Програма2 мають працювати одночасно (спочатку запустити Програму2, потім Програму1).

При цьому обмін повідомленнями відбувається за допомогою текстових файлів. Програма 1 записує повідомлення у файл "msg.txt". Після завершення запису, створює файл "1.txt", у єдиний рядок якого записує кількість повідомлень. Кількість підготовлених повідомлень має також бути показана на екрані. Далі Програма1 очікує, поки Програма2 обробить усі повідомлення.

Кожне повідомлення – це рядок, який має вигляд:

```
call,<ім'я методу>,<параметр1>,...,<параметр n>
```

Програма2 очікує, коли буде готовий файл з повідомленнями (коли файл “1.txt” стане непорожнім).

Цикл очікування можна реалізувати за допомогою функції sleep з модуля time

```
import time
```

```
...
```

```
while True:
```

```
    try:
```

```
        with open("1.txt", "r") as f:
```

```
            s = f.read()
```

```
            if s:
```

```
                break
```

```
        except IOError:
```

```
            pass
```

```
    time.sleep(1)
```

Після того, як файл “msg.txt” готовий, Програма2 робить файл “1.txt” порожнім, читає повідомлення з файлу “msg.txt”, перевіряє, чи є відповідні методи у класі Reserpiant та викликає потрібний метод. Усі методи повинні просто показувати значення своїх параметрів. Усі параметри є рядками. Якщо методу немає у класі, Програма2 показує повідомлення про помилку, але продовжує працювати.

Методи класу Reserpiant можуть додаватись у динаміці. Для цього використовують спеціальне повідомлення

```
add_method,<ім'я методу>,<кількість параметрів>.
```

Отримавши таке повідомлення, Програма2 має додати метод, що показує значення своїх параметрів, до об'єкту класу Reserpiant та в подальшому може отримувати повідомлення для виклику цього методу.

Після того, як Програма2 розібрала усі повідомлення, вона створює файл “2.txt”, у єдиний рядок якого записує кількість оброблених повідомлень. Кількість оброблених повідомлень має також бути показана на екрані.

Програма1 чекає, поки файл “2.txt” стане непорожнім, робить його порожнім, та знову створює файл “msg.txt” з новими повідомленнями для Програми2.

Перевірити, чи є потрібний метод, можна за допомогою стандартної функції getattr, а додати новий метод – за допомогою стандартної функції setattr.

## Необхідні передумови

Засвоєння Теми 19 «Метакласи та метапрограмування».

## Мета завдання

Навчитись писати програми, що використовують метапрограмування у Python.

## Обмеження на виконання завдання

Немає.

## Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо описати Програму 1 та Програму 2.

## Зауваження щодо вирішення завдання

Це завдання допомагає студентам засвоїти початкові засади паралельного програмування, програмування у мережі (без самої мережі) а також використання метапрограмування.

## Рекомендації щодо перевірки результатів

Наочно перевірити правильність зображення а також впевнитись у реалізації описаних класів та функцій.

## Текст програми з розв'язками

Програма 1

```
import time

def wait(in_file):
    while True:
        try:
            with open(in_file, 'r') as f:
                s = f.read()
            if s:
                break
        except IOError:
            pass
        time.sleep(1)
    with open(in_file, 'w') as g:
        pass
    return int(s)

def generate(msg_file, out_file):
    print('program 1: generating messages')
    with open(msg_file, 'w') as f:
        num = 0
        while True:
            msg = input('program1: message:')
            if not msg:
```

```

        break
    msg = msg.replace(' ', '')
    f.write(msg + '\n')
    num += 1

with open(out_file, 'w') as g:
    g.write('{}\n'.format(num))

if __name__ == '__main__':
    print('Program 1 starting')
    while True:
        generate("msg.txt", "1.txt")
        wait("2.txt")

```

Програма 2

```

import time

class Receptient:
    def f1(self, s1, s2):
        print(s1, s2)

    def f2(self, s1, s2, s3):
        print(s1, s2, s3)

def new_meth(params_num):
    def f(*args):
        if len(args) != params_num:
            raise TypeError("Waited for {} params, but got {}".format(params_num, len(args)))
        print(*args)

    return f

def wait(in_file):
    while True:
        try:
            with open(in_file, 'r') as f:
                s = f.read()
            if s:
                break
        except IOError:
            pass

```



```

        time.sleep(1)
    with open(in_file, 'w') as g:
        pass
    return int(s)

def process_message(msg, rc):
    if not msg:
        raise ValueError("Empty message")

    parts = msg.split(',')
    if len(parts) < 3:
        raise ValueError(
            "Message must have at least 3 parts: "
            "(command, name, param(s))")

    command = parts[0]
    meth_name = parts[1]
    params = parts[2:]
    if command == 'call':
        meth = getattr(rc, meth_name, None)
        if not meth:
            raise AttributeError(
                "No method with name {}".format(meth_name))

        meth(*params)
    elif command == 'add_method':
        setattr(rc, meth_name, new_meth(int(params[0])))
    else:
        raise ValueError(
            "Invalid message command {}".format(command))

def process(msg_file, out_file, rc, num):
    print('program 2: processing {} messages'.format(num))
    with open(msg_file, 'r') as f:
        for i in range(num):
            s = f.readline()
            if not s:
                raise ValueError("Invalid meassages number")

            msg = s.strip()
            print('program 2: got message: {}'.format(msg))
            try:
                process_message(msg, rc)
            except Exception as e:
                print(e)

```

```

with open(out_file, 'w') as g:
    g.write('{}\n'.format(num))

if __name__ == '__main__':
    print('Program 2 starting')
    rc = Receptient()
    while True:
        num = wait("1.txt")
        try:
            process("msg.txt", "2.txt", rc, num)
        except Exception as e:
            print(e)

```

## Групове завдання 26. Введення даних форми

### Текст завдання

Скласти програму з графічним інтерфейсом для введення даних деякої форми.

Форма описана у текстовому файлі як послідовність рядків:

<назва> <поле> ,

Де назва – назва поля форми, а поле – вказання типу поля.

Наприклад,

Прізвище {}

Поле може мати один з двох типів:

- Поле введення
- Список

Поле введення позначається фігурними дужками {}

Список позначається квадратними дужками [], у яких через кому вказують елементи списку.

Наприклад,

Стать [чол,жін].

Форма призначена для введення та збереження низки записів з однаковим набором полів.

Форма має розміщуватись на екрані у одному вікні. Окрім надписів та полів введення (списків), у вікні мають також бути кнопки: «Далі», «Готово», «Відмінити».

- Кнопка «Далі» - зберегти дані у файлі, очистити всі поля, продовжити введення
- Кнопка «Готово» - зберегти дані у файлі, завершити введення,

- Кнопка «Відмінити» . не зберігати дані у файлі, очистити всі поля, продовжити введення

Вважати, що всі дані форми можуть розміститись на екрані.

Введені дані зберігати у текстовому файлі, для однієї форми – один рядок. Елементи даних брати у лапки, між елементами – кома.

Наприклад:

“Іваненко”, “чол”

Після введення та збереження усіх даних у файлі (натиснуто кнопку «Готово») показати ці дані у окремому вікні у списку рядків.

Опис форми містяться у окремому файлі form.txt.

Для розв’язання задачі описати 2 класи: FormConstructor та FormsGUI.

FormConstructor будує форму за її описом у файлі, забезпечує введення даних та їх збереження у файлі.

FormsGUI організовує введення, відкриває вікно з інтерфейсом FormConstructor, показує введені дані у списку.

Для того, щоб одночасно у вікні працювати з декількома списками та зберігати вибір у кожному з них, треба задати для кожного списку параметр exportselection=0. Наприклад,

```
lb=Listbox(top, exportselection=0)
```

Подія вибору елементу зі списку - <<ListboxSelect>>. Зв’язати з цією подією функцію обробки some\_handler можна так:

```
lb.bind('<<ListboxSelect>>', some_handler)
```

### Необхідні передумови

Засвоєння Теми 20 «Графічний інтерфейс».

### Мета завдання

Навчитись писати програми, що використовують графічний інтерфейс у Python.

### Обмеження на виконання завдання

Немає.

### Пропозиції щодо розпаралелювання роботи над завданням у команді

Окремо описати класи FormConstructor та FormsGUI.

### Зауваження щодо вирішення завдання

Окрім завдання переслати студентам опис форми для введення.

Для скорочення часу розробки програми можна взяти в якості прототипу FormConstructor клас DictEditor – редактор словників, який є прикладом у матеріалах лекцій до теми «Графічний інтерфейс»

## Рекомендації щодо перевірки результатів

Наочно перевірити правильність зображення а також впевнитись у реалізації описаних класів та функцій.

## Текст програми з розв'язками

Модуль form\_constructor

*# Клас конструктор форм*

```
from tkinter import *
```

```
class FormConstructor:
```

```
    '''
```

```
    Клас призначено для створення форми у графічному режимі  
    за описом у файлі.
```

```
    self.master - вікно, у якому розміщується вікно редагування.
```

```
    self.filename - ім'я файлу з описом форми
```

```
    self.out_file - ім'я файлу для збереження результатів  
введення
```

```
    self.elements - список рядків файлу з описами полів
```

```
    self.has_buttons - чи є власні кнопки у вікна редагування
```

```
    self.vars - список з текстовими змінними для зв'язування
```

```
                з полями введення та списками
```

```
    self.labels - список з надписами
```

```
    self.ent_lists - список з полями введення або списками
```

```
    '''
```

```
    def __init__(self, master, filename, out_file,  
has_buttons=True):  
        self.master = master  
        self.filename = filename  
        with open(self.filename, 'r', encoding='utf-8') as f:  
            self.elements = f.readlines()  
            self.elements = list(map(lambda x: x.strip(),  
self.elements))
```

```
        self.out_file = out_file
```

```
        self.has_buttons = has_buttons
```

```
        self._make_widgets()
```

```
    def _make_widgets(self):
```

```
        '''Створити елементи інтерфейсу форми.'''
```

```
        # рамка для полів введення та списків
```

```
        self.fedit = Frame(self.master, bd=1, relief=SUNKEN)
```

```
        self._make_entries()
```

```
        self._layout_entries()
```

```
        if self.has_buttons:
```

```
            fbut = Frame(self.master)    # рамка з кнопками
```

```

fbut.grid(row=1, column=0, sticky=(E, W))
# кнопка 'Відмінити'
bcancel = Button(fbut, text = 'Відмінити',
                  command = self.cancel_handler)
bcancel.grid(row=0, column=2, sticky=(E), padx=5,
pady=5)

# кнопка 'Готово'
bok = Button(fbut, text = 'Готово',
              command = self.ok_handler)
bok.grid(row=0, column=1, sticky=(E), padx=5, pady=5)
# кнопка 'Далі'
bnext = Button(fbut, text = 'Далі',
                command = self.next_handler)
bnext.grid(row=0, column=0, sticky=(E), padx=5,
pady=5)

# забезпечити зміну розмірів області кнопок
fbut.columnconfigure(0, weight=1)

def _get_label_type_values(self, line):
    if line[-1] == '}': # поле введення
        label = line.split('{')[0].strip()
        typ = "entry"
        values = []
    else: # список
        parts = line.split('[')
        label = parts[0].strip()
        typ = "list"
        values_str = parts[1].strip(']')
        values = values_str.split(',')
    return label, typ, values

def _make_entries(self):
    '''Створити надписи та поля введення або списки.'''
    self.vars = []
    self.labels = []
    self.ent_lists = []
    for i, line in enumerate(self.elements):
        label, typ, values =
self._get_label_type_values(line)
        # додати надпис до словника надписів
        self.labels.append(Label(self.fedit, text=label))
        # створити текстову змінну для поля введення або
списку
        # та встановити її початкове значення
        self.vars.append(StringVar())
        self.vars[-1].set("")
        # додати поле введення або список та зв'язати з

```

ТЕКСТОВОЮ ЗМІННОЮ

```
        if typ == "entry":
            self.ent_lists.append(
                Entry(self.fedit, textvariable=self.vars[-
1]))

        else:
            self.ent_lists.append(
                Listbox(self.fedit, exportselection=0,
                        width=len(max(values, key=len)),
                        height=len(values)))
            self.ent_lists[-1].bind('<<ListboxSelect>>',
self.select_handler(i))
            for val in values:
                self.ent_lists[-1].insert(END, val)

def _layout_entries(self):
    '''Розмістити надписи та поля введення або списки.'''
    for i in range(len(self.labels)):
        # розмістити надписи у першому стовпчику
        self.labels[i].grid(row=i, column=0,
                            sticky=(W), padx=1, pady=1)
        # розмістити поля введення у другому стовпчику
        self.ent_lists[i].grid(row=i, column=1,
                                sticky=(W, E), padx=1,
pady=1)
        # розташувати рамку у вікні self.master
        self.fedit.grid(row=0, column=0, sticky=(W,E,N,S))
        # забезпечити зміну розмірів рамок з елементами та
кнопками
        self.master.columnconfigure(0, weight=1)
        # забезпечити зміну розмірів області елементів
        self.fedit.columnconfigure(0, weight=1)
        self.fedit.columnconfigure(1, weight=2)

def select_handler(self, i):
    '''Обробити вибір елементу зі списку.'''
    def handle(ev):
        self.vars[i].set(self.ent_lists[i].get(
            self.ent_lists[i].curselection()))

    return handle

def ok_handler(self, ev=None):
    '''Обробити натиснення кнопки "Готово".'''
    self._save()
    self.master.destroy()    # закрити вікно self.master
```

```

def next_handler(self, ev=None):
    '''Обробити натиснення кнопки "Далі".'''
    self._save()
    self._clear()

def cancel_handler(self, ev=None):
    '''Обробити натиснення кнопки "Відмінити".'''
    self._clear()

def _clear(self):
    for v in self.vars:
        v.set("")

def _save(self):
    with open(self.out_file, 'a', encoding='utf-8') as f:
        parts = ["{}"].format(x.get()) for x in self.vars]
        line = ','.join(parts) + '\n'
        f.write(line)

def get(self):
    '''Повернути останні значення усіх полів введення або списків.'''
    return [v.get() for v in self.vars]

if __name__ == '__main__':
    top = Tk()
    fc = FormConstructor(top, 'form.txt', 'data.txt')
    top.mainloop()
    d = fc.get()
    print(d)

```

Модуль forms\_gui

```

from tkinter import *
from form_constructor import FormConstructor

DATA_FILE = "data.txt"
FORM_FILE = "form.txt"

class FormsGUI:
    '''Клас для організації введення даних за допомогою форми.

    self.top - вікно верхнього рівня у якому розміщено
    елементи
    інтерфейсу
    self.data_file - ім'я файлу з даними

```

```

        self.form_file - ім'я файлу опису форми
'''

def __init__(self, master, form_file, data_file):
    self.top = master
    self.data_file = data_file
    self.form_file = form_file
    with open(self.data_file, 'w'): # очистити файл
        pass
    self._make_widgets()

def _make_widgets(self):
    '''Створити елементи інтерфейсу.'''
    self.finput = Frame(self.top) # контейнер для списку з
даними
    self.finput.pack(fill=X, expand=YES)
    self.sb_all = Scrollbar(self.finput)
    self.sb_all.pack(side=RIGHT, fill=Y)
    self.l_all = Listbox(self.finput, height=15, width=70,
                        yscrollcommand=self.sb_all.set,
                        font=('arial', 16))
    self.sb_all.config(command=self.l_all.yview)
    self.l_all.pack(side=RIGHT, fill=BOTH, expand=YES)

    self.fbut = Frame(self.top) # контейнер для кнопок
    self.fbut.pack(side=LEFT, fill=X, expand='1')
    self.benter = Button(self.fbut, text='Ввести дані',
                        command=self._enter_data,
                        font=('arial', 16))
    self.benter.pack(side=LEFT, padx=5, pady=5)
    self.bquit = Button(self.fbut, text='Закрити',
                        command=top.quit,
                        font=('arial', 16))
    self.bquit.pack(side=RIGHT, padx=5, pady=5) # кнопка
"Закрити"

def _enter_data(self):
    '''Ввести дані у форму'''
    dialog = Toplevel()
    dl = FormConstructor(dialog, self.form_file,
self.data_file)
    # зробити діалог модальним
    dialog.focus_set()
    dialog.grab_set()
    dialog.wait_window()
    self._fill_list()

```



```

def _fill_list(self):
    '''Заповнити список.'''
    self.l_all.delete(0, END)
    with open(self.data_file, 'r', encoding='utf-8') as f:
        for line in f:
            self.l_all.insert(END, line.strip())

if __name__ == '__main__':
    from sys import argv

    if len(argv) < 3:
        form_file = FORM_FILE
        data_file = DATA_FILE
    else:
        form_file = argv[1]
        data_file = argv[2]
    top = Tk()
    r = FormsGUI(top, form_file, data_file)
    mainloop()

```

## Список літератури

1. Обвінцев О.В. Інформатика та програмування. Курс на основі Python. Матеріали лекцій. – К., Основа, 2017
2. A Byte of Python (Russian) Версия 2.01 Swaroop С Н (Translated by Vladimir Smolyar), <http://wombat.org.ua/AByteOfPython/AByteofPythonRussian-2.01.pdf>
3. Марк Лутц, Изучаем Python, 4-е издание, 2010, Символ-Плюс
4. Марк Саммерфилд, Программирование на Python 3. Подробное руководство. - Символ-Плюс, 2009.