МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ МОРСЬКИЙ УНІВЕРСИТЕТ

Кафедра «Технічна кібернетика й інформаційні технології ім. професора Р.В. Меркта»

Затверджено
НМК ННІ інформаційних технологій та інноваційного підприємництва
Протокол №4 від 27.12.2023 р.
Керівник НМК ННІ ІТІП

Андрій ІВАНОВ
«27» грудня 2023 р.

Конспект лекцій

до курсу

«ОСНОВИ ПРОГРАМУВАННЯ НА МОВІ PYTHON»

(дисципліна вільного вибору студентів з каталогу дисциплін ОНМУ)

Одеса – 2023

Конспект лекцій з дисципліни «Основи програмування на мові Python» підготовлений кандидатом фізико-математичних наук ***Розум Мариною Валеріївною*** – доцентом кафедри «Технічна кібернетика й інформаційні технології імені проф. Р.В. Меркта» Одеського національного морського університету.

Конспект лекцій з дисципліни «Основи програмування на мові Python» схвалено кафедрою «Технічна кібернетика й інформаційні технології імені проф. Р.В. Меркта» ОНМУ (протокол №8 від « 22 » грудня 2023 р.)

Рецензент – Малаксіано М.О., д.т.н., професор, зав. кафедрою КТКіІТ імені проф. Р.В. Меркта

ЗМІСТ

ВВЕДЕННЯ

Дисципліна «Основи програмування на мові Python» спрямована на формування у здобувачів вищої освіти теоретичних основ та практичних навичок застосування мови програмування Python, її бібліотек та відповідних програмних інструментів для створення сучасних програмних продуктів та проведення наукових розрахунків під час вирішення прикладних задач в області комп'ютерних наук.

Метою навчальної дисципліни є формування теоретичних знань і набуття навичок самостійної розробки програм на мові Python для розробки сучасних програмних і веб-додатків, проведення наукових розрахунків, обробки даних та візуалізації результатів.

У результаті вивчення дисципліни здобувачі повинні здобути знання базового синтаксису мови програмування Python; знання основних структур даних, елементів програми; отримати основні відомості про використання бібліотек мови Python.

Кожна лекція ілюструється багатьма прикладами, які подано з аргументованими поясненнями. Для закріплення знання, отриманого в теоретичній частині, пропонується вирішення задач, які надано в методичних вказівках щодо проведення практичних занять з дисципліни «Основи програмування на мові Python», укладач М.В. Розум.

Лекція 1. Введення. Знайомство з Python

Анотація. Що таке програма та які існують мови програмування? Чим хороша мова Python? Як встановити на комп'ютер інтерпретатор Python та середовище розробки PyCharm? Які бувають помилки під час написання коду програми та як їх виправити? Введення та виведення даних у мові Python. Нескладні програми, які можуть щось виводити на екран (команда ***print()***) і зчитувати інформацію з клавіатури (команда ***input()***). Необов'язкові параметри команди ***print()*** та детальніший розбір поняття змінних. Коментарі та стандарт PEP 8, якого дотримуються Python-програмісти. Робота з цілими числами. Основні операції, які застосовуються з цилими числами, перетворювання рядків на числа. Додаткові операції під час роботи з цілими числами. Обробка цифр цілого числа.

План:

1. Історія мови Python
2. Сильні та слабкі сторони Python
3. Python 2 VS Python 3
4. Встановлення Python на комп'ютер
5. Встановлення PyCharm на комп'ютер
6. Помилки під час написання коду
7. Виведення даних, команда `print()`
8. Введення даних, команда `input()`
9. Розв'язання задач
10. Необов'язкові параметри команди print(): sep, end
11. Змінні
12. Стандарт PEP 8
13. Коментарі
14. Розв'язання задач
15. Цілочисленний тип даних
16. Перетворення рядка до цілого числа
17. Операції над цілими числами
18. Розв'язання задач
19. Операція зведення у ступінь
20. Операція знаходження залишку
21. Операція цілісного поділу
22. Обробка цифр числа
23. Розв'язання задач

Комп'ютерна програма – список команд (інструкцій) для комп'ютера. Команди можуть бути будь-якими, наприклад:

• зчитати інформацію з клавіатури;
• провести арифметичні обчислення (+, −, *, /);
• вивести інформацію на екран.

У кожному комп'ютері встановлено багато різноманітних програм. Наприклад, Google Chrome, - це програма-браузер. Вона дає змогу

переглядати сторінки сайтів в інтернеті. Програма Telegram або Viber дозволяє здійснювати дзвінки та обмінюватися миттєвими повідомленнями. Зрештою, сама операційна система, чи то Windows, OS X або Linux, також програма.

Для створення програм використовують мови програмування. Вибір мови програмування, зазвичай, продиктований особливостями самої програми.

Мова програмування - набір певних правил, за якими комп'ютер може розуміти команди (інструкції) та виконувати їх. Текст програми будь-якою мовою програмування, називається ***програмним кодом***.

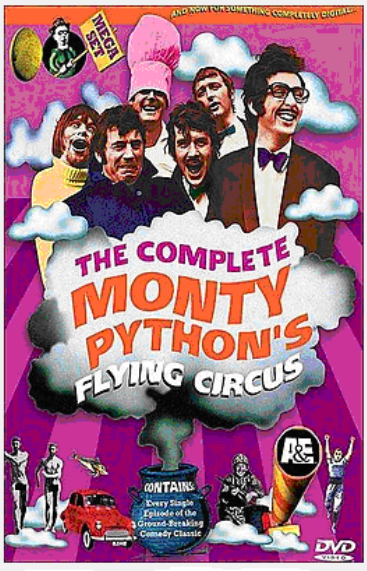
Мови програмування бувають ***компілювані*** та ***інтерпретовані***. Якщо програма написана компілюємою мовою (C, C++, Pascal), перед виконанням її потрібно повністю перевірити наявність синтаксичних помилок і вже після цього перекласти у зрозумілу комп'ютера форму — ***машинний код***. Це робить спеціальна програма, яка називається ***компілятором***.

Якщо програма написана мовою, що інтерпретується (Python, PHP, Ruby), вона не перекладається в машинний код повністю. Натомість спеціальна програма, яка називається інтерпретатором, йде за кодом, аналізує його та виконує кожну окрему команду.

Існують мови програмування, які поєднують обидва підходи (C#, Java). У таких мовах код вихідної програми спочатку компілюється в проміжний код (***байт-код***), а потім, під час виконання, перекладається в машинний код.

Мову ***Python*** розробив голландський програміст ***Гвідо Ван Россум*** (Guido van Rossum) у 1991 році. Гвідо був фанатом британського комедійного серіалу "Monty Python's Flying Circus", звідки і прийшла назва мови.

Гвідо Ван Россум



Monty Python's Flying Circus

Python 2 VS Python 3. Основні версії мови Python – ***Python 2*** та ***Python 3***. Версія ***Python 2*** вважається застарілою, версія 3 – новішою та сучаснішою. Чому не відмовляться від другої версії? Якщо коротко, ***Python 3*** не має повної зворотної сумісності з попередньою версією: на ***Python 2*** написано дуже багато програм, і розробники не мають можливості переписати все на нову версію. У нашому курсі ми будемо користуватися тільки ***Python 3*** і не говоритимемо про ***Python 2***.

Переваги Python

1. Це інтерпретована мова програмування:
   - вона не потребує окремого етапу компіляції;
   - програма на мові Python запускається прямо з вихідного коду;
2. Це високорівнева мова програмування;
3. Це платформонезалежна мова:
   - програми на Python можна ***створювати*** на різних операційних системах (Linux, Windows, OS X);
   - програми Python можна ***запускати*** на різних операційних системах (Linux, Windows, OS X);
4. Це ***open source проект***;
5. Це проста мова;
6. Це ***вбудована скриптова мова***;
7. Це ***динамічна мова***, що спрощує написання нескладних програм;
8. Для Python існує велика бібліотека класів на будь-який смак.

Недоліки Python

1. ***Низька швидкість виконання*** порівняно з такими мовами, як C та C++;
2. Динамічна типізація мови – мінус під час написання складних програм.

Задачі, які вирішуються за допомогою Python

Python підходить для вирішення широкого спектру завдань. Розіб'ємо їх на категорії:

1. ***Системне програмування***. Вбудовані в Python інтерфейси доступу до служб операційних систем роблять його ідеальним інструментом для створення програм, що переносяться, і утиліт системного адміністрування;
2. ***Графічні додатки***. Простота Python та швидкість розробки роблять його чудовим засобом створення графічного інтерфейсу. До складу Python входить стандартний об'єктно-орієнтований інтерфейс до GUI API;
3. ***Веб-додатки***. За допомогою додаткових фреймворків на мові Python (Django, Flask, Pyramid) можна створювати повнофункціональні сайти;
4. ***Веб-сценарії***. Python поставляється разом із стандартними інтернет-модулями, які дозволяють програмам виконувати різноманітні мережеві операції як у режимі клієнта, так і в режимі сервера;

5. ***Інтеграція компонентів***. Можливість Python розширюватися і вбудовуватись у системи мовою C++ робить його зручним для опису поведінки інших систем та компонентів;
6. ***Програми баз даних***. У Python є інтерфейси доступу до всіх основних реляційних баз даних: Sybase, Oracle, Informix, ODBC, MySQL, PostgreSQL, SQLite та багатьох інших. З їх допомогою можна створювати програми баз даних.

Проекти, у яких використовується Python

1. Компанія Google використовує Python у своїй пошуковій системі;
2. Компанії Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm та IBM використовують Python для тестування апаратного забезпечення;
3. Сервіс YouTube значною мірою реалізований на Python;
4. Агентство національної безпеки (NSA) використовує Python для шифрування та аналізу даних;
5. Компанії JPMorgan Chase, UBS, Getco та Citadel застосовують Python для прогнозування фінансового ринку;
6. Програма BitTorrent для обміну файлами у піринговых мережах написана мовою Python;
7. NASA, Los Alamos, JPL та Fermilab використовують Python для наукових обчислень.

Крім того, Python використовувався у розробці:

- Blender (програма для створення 3D-графіки);
- Ubuntu Software Center (центр програм в ОС Ubuntu);
- BitTorrent, до 6-ї версії (менеджер торрент-закачування);
- World of Tanks (популярна гра).

Філософія Python

Розробники мови Python дотримуються певної філософії програмування, що називається The Zen of Python. Її текст видається інтерпретатором Python за командою `import this` (працює один раз за сесію). Автором цієї філософії вважається Тім Петерс (Tim Peters).

В оригіналі

1. Beautiful is better than ugly;
2. Explicit is better than implicit;
3. Simple is better than complex;
4. Complex is better than complicated;
5. Flat is better than nested;
6. Sparse is better than dense;
7. Readability counts;

8. Special cases aren't special enough to break the rules;
9. Although practicality beats purity;
10. Errors should never pass silently;
11. Unless explicitly silenced;
12. В face of ambiguity, refuse the temptation to guess;
13. There should be one – and preferably only one – obvious way to do it;
14. Although, що може не бути впевненим у першому неналежному вам Dutch;
15. Now is better than never;
16. Although never is often better than *right* now;
17. Якщо implementation is hard to explain, it's a bad idea;
18. Якщо implementation is easy to explain, it may be a good idea;
19. Namespaces are one honking great idea — 's's more of those!

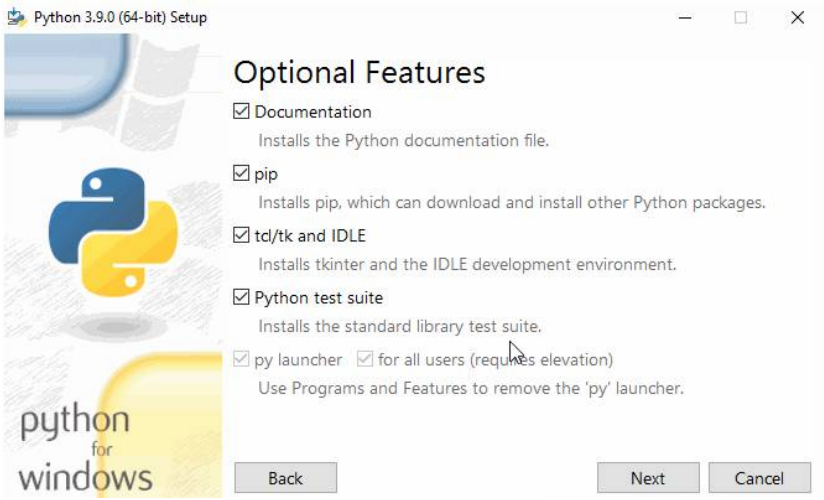
Встановлення Python на Windows

Мова Python відноситься до вільного програмного забезпечення, тому його можна завантажити з офіційного сайту, вільно розповсюджувати та встановлювати на всі сучасні операційні системи.

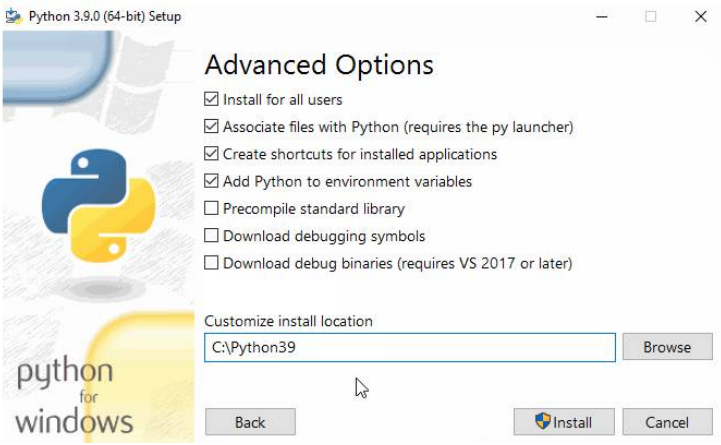
Встановлення **Python 3.9** (64-bit) для **Windows**

1. З’ясуйте розрядність вашої операційної системи.
2. Перейдіть на сайт https://www.python.org/downloads/ .
3. Оберіть версію Python.
4. Завантажте файл з розширенням .exe відповідної розрядності.
5. Встановіть Python:
   a. відзначте рекомендований параметр `Install launcher for all users`
   b. не забудьте встановити прапорець `Add Python 3.x to PATH` (це полегшить правильне налаштування системи)

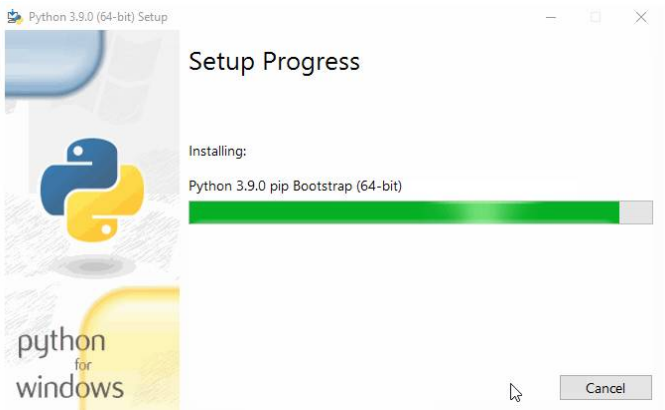
c. оберіть варіант налаштування установки `Customize installation`

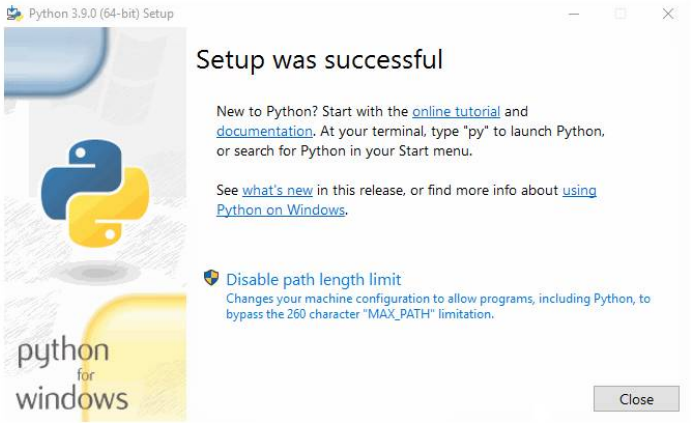


d. вкажіть каталог установки `C:\PythonX` (де `X` - номер версії)



Нажміть на ***Install***





Перевіримо, чи Python успішно був встановлений на комп'ютер. Для цього натисніть сполучення клавіш `Win+R` на клавіатурі, введіть команду `cmd` і натисніть OK. У консольному вікні, що з'явилося, введіть команду `python --version` і натисніть `Enter`:

```
> python --version
Python 3.9.0
```

Якщо ви отримали схожий результат, то `Python` відповідної версії успішно встановлений у вашій системі.

На момент створення курсу останньою версією є Python 3.9.0.

Середовище розробки `PyCharm`

Команди для інтерпретатора можна писати у звичайному текстовому редакторі (наприклад, "Блокноті"). Але найчастіше для цього користуються спеціальною програмою, яка називається середовищем розробки (англ. IDE, Integrated Development Environment).

Середовище розробки теж текстовий редактор, але з додатковими можливостями. Наприклад, воно вміє самостійно знаходити на комп'ютері програму-інтерпретатор та запускати програму однією кнопкою. Середовище розробки, крім того, форматує написаний вами код, щоб його зручно було читати, а іноді підказує, де ви припустилися помилки.

Ми будемо використовувати інтегроване середовище розробки PyCharm. `PyCharm` є власницьким програмним забезпеченням. Наявна безкоштовна версія `Community` з усіченим набором можливостей. Для користувачів `Linux`, `Windows` і `macOS`.

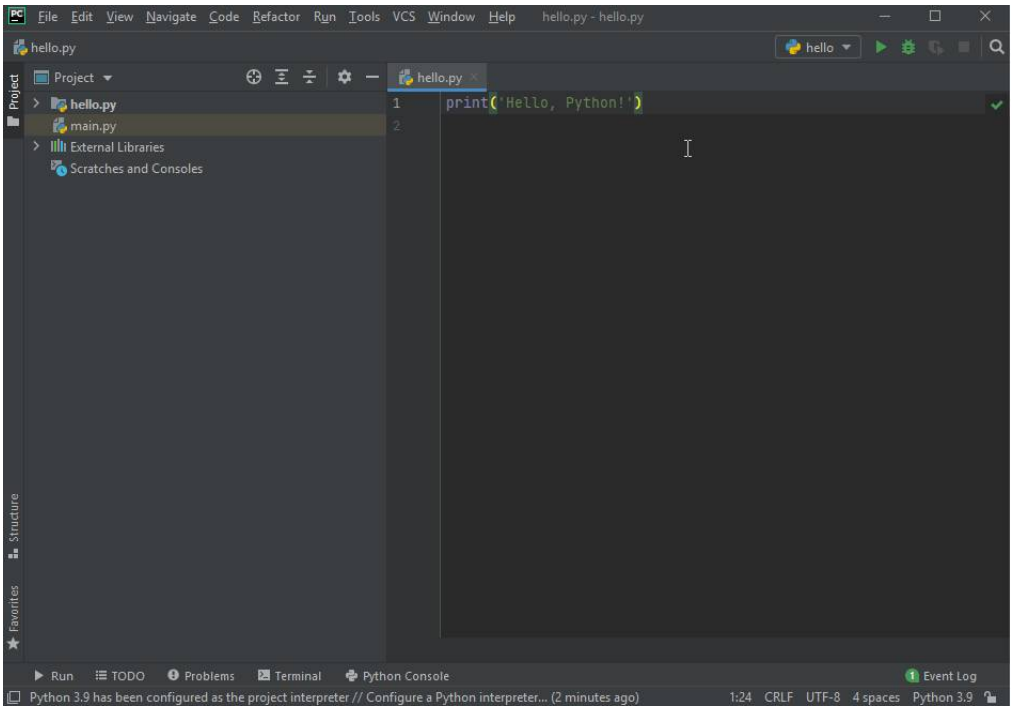
https://www.jetbrains.com/pycharm/download/#section=windows

Для написання програм мовою `Python` можна використовувати онлайн-середовище repl.it .

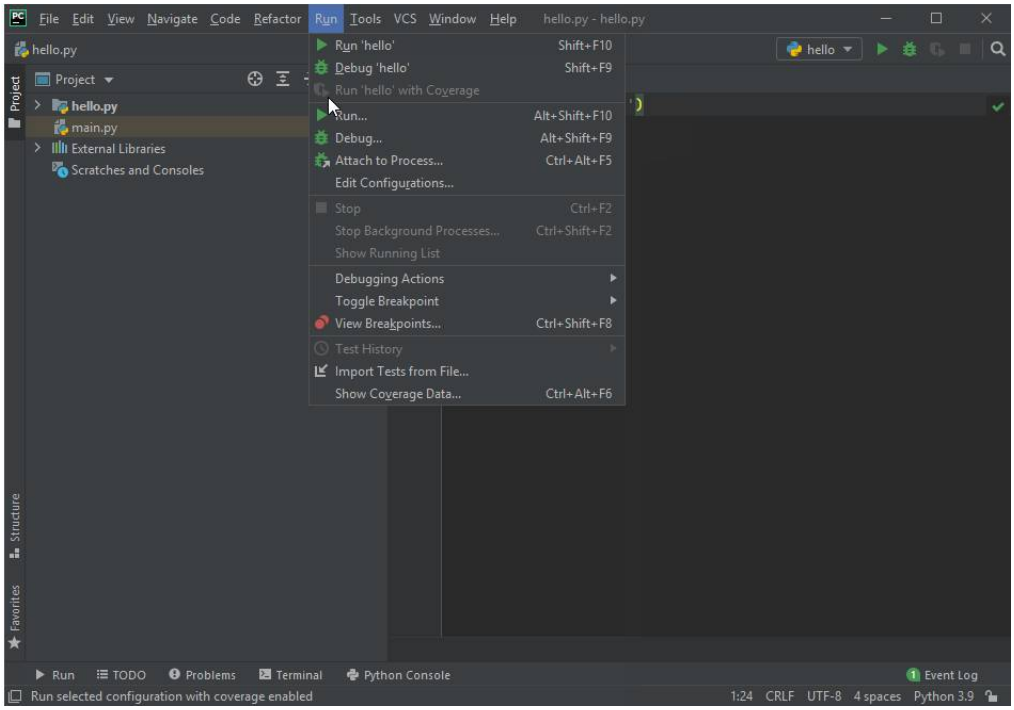
Для встановлення, запуску і налаштування `PyCharm` використовуйте інструкцію (англ.) на офіційному сайті.
https://www.jetbrains.com/help/pycharm/installation-guide.html

PyCharm для Python у Windows

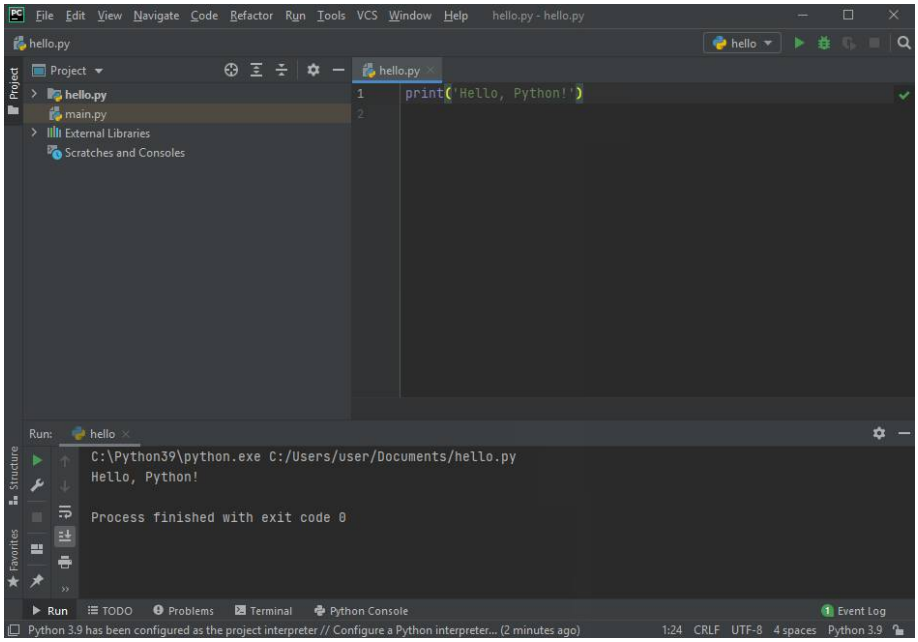
1 крок – створили файл ***hello.py***



2 крок – запуск файла



3 крок – перегляд результату



У режимі **інтерактивного інтерпретатора** команди вводяться у консольному вікні одна за одною і по натисненні клавіші `Enter` відразу виконуються з відображенням результату виконання.

Перейдіть у вікно Python Console.
Якщо на екрані з'явиться запрошення `>>>` до введення команд, значить система працює в інтерактивному режимі.

Повідомлення про помилку

В процесі написання і виконання програм можуть з'являтися різноманітні помилки. У таких випадках інтерпретатор Python сам сигналізує про помилку.

Наприклад, коли ми введемо в режимі інтерактивного інтерпретатора інструкцію '19' + 81, з'явиться таке повідомлення:

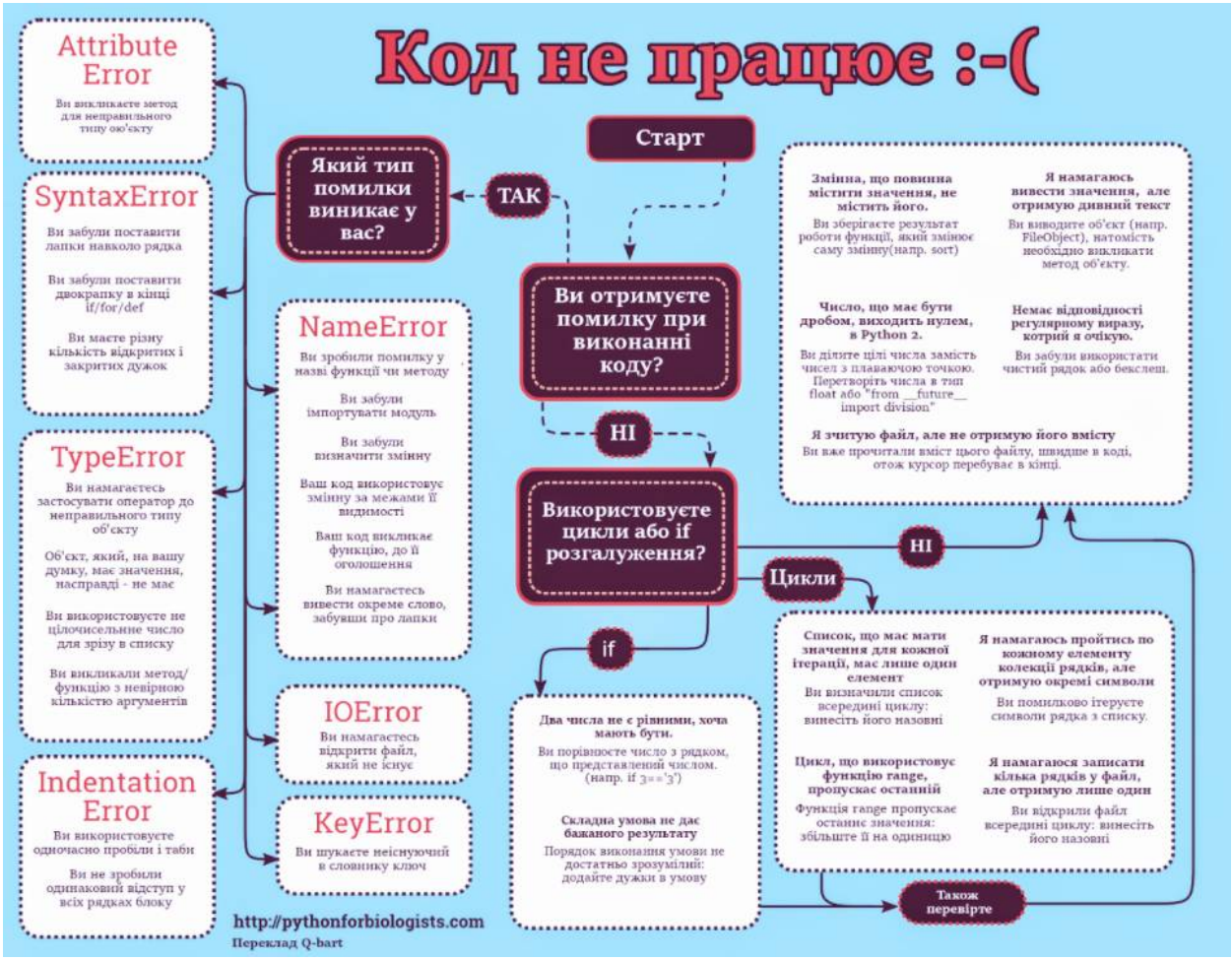
```
>>> '19' + 81
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Введена інструкція некоректна для `Python`, тому він вказав назву помилки і номер рядка, в якому вона виникла, зупинивши виконання програми.

*У `Python`, у разі появи помилки **генерується виняток**, який повідомляє про зміст помилки. В даному випадку, згенерований виняток `TypeError` повідомляє про несумісність типів під час додавання числа і рядка, іншими словами, рядок можна об'єднувати лише з рядком.*

Якщо помилка зрозуміла для нас, її виправляють. В іншому випадку, щоб дізнатися, що означає повідомлення про помилку, можна здійснити пошук в мережі Інтернет за назвою помилки.

Основні помилки, що можуть виникнути при написанні програм на `Python` *і шляхи їх виправлення*



Введення-виведення даних

Виведення даних, команда print()

Для виведення даних на екран використовується команда `print()`.

Усередині круглих дужок пишемо, що хочемо вивести на екран. Якщо це текст, обов'язково вказуємо його всередині лапок. Лапки можуть бути одинарними або подвійними. Тільки обов'язково ставимо однакові до та після тексту.

Наприклад, наступний код:

```
print('Ми вивчаємо мову Python')
```

виведе на екран текст:

```
Ми вивчаємо мову Python
```

```
>>> print('Ми вивчаємо мову Python')
Ми вивчаємо мову Python
```

Запам'ятай: лапки можуть бути і одинарними, і подвійними. Наступні два рядки роблять те саме.

```
print('Python')
print("Python")
```

Те, що ми пишемо у круглих дужках у команди `print()`, називається ***аргументами*** чи ***параметрами*** команди.
Команда `print()` дозволяє вказувати кілька аргументів, у такому разі їх треба відокремлювати комами.
Наприклад, наступний код:

```
>>> print('Скоро я', 'програмуватиму', 'мовою', 'Python!')
```

виведе на екран текст:

Скоро я програмуватиму мовою Python!

Зверніть увагу, як роздільник при виведенні даних між аргументами команди використовується ***символ пробілу***. За замовчуванням команда `print()` додає рівно один пробіл між усіма своїми аргументами. Наприклад, наступна програма:

```
print('1', '2', '4', '8', '16')
```

виводить числа 1 2 4 8 16. Числа виведені рівно через один пробіл.
Запам'ятай: при написанні коду між аргументами команди `print()` після коми ми ставимо 1 символ пробіл. Це загальноприйнята угода у мові Python. Цей символ пробілу не впливає на виведення даних.

Примітки

Примітка 1. Команда `print()` записується лише маленькими літерами, інше написання неприпустимо, оскільки в Python малі і великі літери ***різні***.
Примітка 2. Кожна наступна команда `print()` виводить цей текст з ***нового рядка***. Наприклад, наступна програма:

```
print('Який гарний день!')
print('Працювати мені не ліньки!')
```

виведе на екран два рядки:

```
Який гарний день!
Працювати мені не ліньки!
```

Примітка 3. Команда `print()` з порожнім списком аргументів просто вставляє новий порожній рядок. Наприклад:

```
print('Який гарний день!')
print()
print('Працювати мені не ліньки!')
```

виведе на екран три рядки, один з яких порожній:

```
Який гарний день!

Працювати мені не ліньки!
```

Примітка 4. Чому в Python можна використовувати як одинарні, так і подвійні лапки для обрамлення тексту? Відповідь дуже проста - часто лапки це частина тексту. І щоб Python міг правильно розпізнати такий текст, користуємося різними:

- якщо в тексті потрібні одинарні лапки, для обрамлення такого тексту використовуємо подвійні лапки;
- якщо в тексті потрібні подвійні лапки, обрамляємо його одинарними.

Результатом виконання коду:

```
print('У тексті є "подвійні" лапки')
print("У тексті є 'одинарні' лапки")
```

буде:

```
У тексті є "подвійні" лапки
У тексті є 'одинарні' лапки
```

Виконання задач 1-4 з практичної роботи 1

Введення даних, команда input()

Усі попередні програми виводили на екран текст, відомий на момент написання програмного коду. Однак програми можуть працювати з даними, які стануть відомими лише під час виконання програми. Іншими словами, програми можуть зчитувати дані, а потім їх використовувати.

Для зчитування даних у мові Python використовується команда `input()`.

Розглянемо таку програму:

```
print('Як тебе звуть?')
name = input()
print('Привіт,', name)
```

Спочатку програма роздрукує текст на екран «Як тебе звуть?». Далі програма чекатиме від користувача введення даних. Введення даних реалізується за допомогою команди `input()`.

Команда `input()` завжди пишеться з круглими дужками. Вона працює так: коли програма доходить до місця, де є `input()`, вона чекає, доки користувач введе текст із клавіатури (введення завершується натисканням клавіші `Enter`). Введений рядок підставляється на місце `input()`.

Тобто якщо ви ввели рядок «Parrot», програма далі працюватиме так, ніби на місці `input()` було написано «Parrot».

Таким чином, `input()` отримує від користувача якісь дані і замість виклику підставляє ***рядкове значення***, в нашому випадку записує його як значення ***змінної*** `name`.

Що таке змінні і що означає зберегти в змінній значення, докладніше поговоримо трохи пізніше, а поки що запам'ятовуємо:

Команда `print()` виводить на екран дані.
Команда `input()` зчитує введені дані з клавіатури.

Примітки

Примітка. Дуже часто перед зчитуванням даних друкуємо деякий текст, щоб користувач, який вводить ці дані розумів, що саме від нього вимагається. Наприклад, у програмі

```
print('Як тебе звуть?')
name = input()
print('Привіт,', name)
```

ми спочатку виведемо текст «Як тебе звуть?», а вже потім зчитаємо дані.

Оскільки це досить поширений сценарій, то в мові Python можна виводити текст, передаючи його як параметр у команду `input()`. Попередній код можна переписати так:

```
name = input('Як тебе звуть?')
print('Привіт,', name)
```

Виконання задач 5-8 з практичної роботи 1

Параметри sep, end, змінні, PEP 8

Необов'язкові параметри команди print

За замовчуванням команда `print()` приймає кілька аргументів (параметрів), виводить їх через один ***пробіл***, після чого ставить ***переведення рядка.*** Цю поведінку можна змінити, використовуючи додаткові іменовані параметри ***sep*** (separator, роздільник) і ***end*** (закінчення).

Параметр sep

Розглянемо наступний код:

```
print('a', 'b', 'c')
print('d', 'e', 'f')
```

Результатом виконання такого коду буде:

```
a b c
d e f
```

Розглянемо наступний код:

```
print('a', 'b', 'c', sep='*')
print('d', 'e', 'f', sep='**')
```

Результатом виконання такого коду буде:

```
a*b*c
d**e**f
```

При першому друку в якості рядка роздільника між аргументами команди `print()` встановлено рядок `sep='*'`.

При другому друку в якості рядка роздільника між аргументами команди `print()` встановлено рядок `sep='**'`.

Таким чином, необов'язковий параметр ***sep*** команди ***print()*** дозволяє встановити рядок, за допомогою якого будуть розділені аргументи під час друку.

Параметр end

Якщо переведення рядка робити не потрібно або потрібно вказати спеціальне закінчення, слід явно вказати значення для параметра ***end***.

Розглянемо наступний код:

```
print('a', 'b', 'c', end='@')
print('d', 'e', 'f', end='@@')
```

Результатом виконання такого коду буде:

```
a b c@d e f@@
```

Після першого друку вставлено рядок @ замість переведення рядка. Аналогічно, після завершення другого друку вставлено рядок @@.

Параметри ***sep*** та ***end*** можна використовувати разом. Розглянемо наступний код:

```
print('a', 'b', 'c', sep='*', end='finish')
print('d', 'e', 'f', sep='**', end='^__^')
print('g', 'h', 'i', sep='+', end='%')
print('j', 'k', 'l', sep='-', end='#')
print('m', 'n', 'o', sep='/', end='!')
```

Результатом виконання такого коду буде:

```
a*b*cfinishd**e**f^__^g+h+i%j-k-l#m/n/o!
```

Примітки

Примітка 1. Виклик команди `print()` із порожніми дужками ставить переведення рядка.

Примітка 2. Послідовність символів `\n` називається керуючою послідовністю і задає переведення рядка.

Примітка 3. Значення за замовчуванням у параметрів `sep` і `end` наступні:

```
sep=' ' # пробіл
end='\n' # переведення рядка
```

Примітка 4. Щоб прибрати всі додаткові символи, можна викликати команду `print()` так:

```
print('a', 'b', 'c', sep='', end='')
```

Примітка 5. Програмний код

```
print('Python')
```

рівнозначний коду

```
print('Python', end='\n')
```

Примітка 6. *Я*кщо після виведення даних потрібно більше одного переведення рядка, необхідно використовувати наступний код:

```
print('Python', end='\n\n\n')
```

Виконання задач 9-11 з практичної роботи 1

Змінні

У попередніх вправах ми вже використовували змінні, але автор курсу досі не пояснив, що це таке.

Ми знаємо, що команда `input()` позначає «почекай, поки користувач введе щось із клавіатури, і запам'ятай те, що він ввів». Просто так просити «запам'ятати» досить безглуздо: адже нам потім треба буде якось сказати комп'ютеру, щоб він згадав те, що запам'ятав. Для цього використовуємо ***змінні*** та пишемо такий код:

```
variable_name = input()
print('Ви ввели текст:', variable_name)
```

Цей код означає: "Збережи те, що ввів користувач, у пам'яті, і далі це місце в пам'яті ми називатимемо ім'ям ***variable_name***".
Відповідно, команда `print(variable_name)` означає: "Подивися, що лежить у пам'яті, під ім'ям ***variable_name***, і виведи це на екран".
Ось такі "місця в пам'яті" називаються змінними. Будь-яка змінна має ***ім'я*** та ***значення.***

Ім'я змінної

1. У імені змінної використовуйте лише латинські літери a-z, A-Z, цифри та символ нижнього підкреслення (_);
2. Ім'я змінної не може починатися з цифри;
3. Ім'я змінної по можливості має відбивати її призначення.

Запам'ятай: Python - регістрочутлива мова. Змінна `name` і `Name` – дві абсолютно різні змінні. Для назви змінних прийнято використовувати стиль ***lower_case_with_underscores*** (слова з маленьких літер з підкресленнями).

Значення змінної

Значення змінної — збережена у ній інформація. Це може бути текст, число і т.д.

Знак «=» це ***оператор присвоювання***. Він надає значення, яке знаходиться праворуч від знака «рівно», змінній, яка знаходиться ліворуч від знака «рівно».

У нашому випадку це те, що помістив до неї користувач командою `input()`. Це текстове значення – рядок. Тобто змінна зберігає у собі рядкове значення. Говорять, що змінна має рядковий тип даних.

Запам'ятай: інтерпретатор чекає, що користувач щось введе з клавіатури рівно стільки разів, скільки команд `input()` зустрічається у програмі. Кожен `input()` завершується натисканням `Enter` на клавіатурі.

Значення змінної, звісно, можна змінювати (перепривласнювати).

```
print('Як тебе звуть?')
name = input()
print('Привіт,', name)
name = 'Марина'
print('Привіт,', name)
```

Оператор присвоювання повідомляє змінній те чи інше значення незалежно від цього, була ця змінна введена раніше. Ви можете змінювати значення змінної, записавши ще один оператор присвоєння. Якщо ми маємо змінну, ми можемо робити з її значенням усе що завгодно — наприклад, привласнити іншій змінній:

```
name1 = 'Марина'
name2 = name1
print(name2)
```

Отже, якщо ви хочете, щоб у вас була ***змінна з якимось ім'ям*** і ***якимось значенням***, потрібно написати на окремому рядку:

<ім'я змінної> = <значення змінної>

Як тільки ця команда виконається, у програмі з'явиться змінна із зазначеним значенням.

Запам'ятай: назва змінної завжди має бути ***ліворуч від знака рівності***.
Наступний код спричинить помилку: 'Maryna' = name.

Множинне присвоєння

У мові Python можна за одну інструкцію присвоєння змінювати значення відразу декількох змінних. Робиться це так:

```
name, surname = 'Maryna', 'Rozum'
print('Ім'я:', name, 'Прізвище:', surname)
```

Цей код можна записати і так:

```
name = 'Maryna'
surname = 'Rozum'
print('Ім'я:', name, 'Прізвище:', surname)
```

Відмінність двох способів у тому, що множинне привласнення першому способі присвоює значення двом змінним одночасно.

Якщо потрібно зчитати текст з клавіатури і присвоїти його в якості значення змінним, можна написати так:

```
name, surname = input(), input()
print('Ім'я:', name, 'Прізвище:', surname)
```

Якщо ліворуч від знака «рівно» у множинному присвоєнні повинні стояти через кому імена змінних, то праворуч можуть стояти довільні вирази, розділені комами. Головне, щоб ліворуч і праворуч від знака присвоювання було однакове число елементів.

Множинне присвоєння зручно використовувати, коли потрібно обміняти значення двох змінних. У Python це робиться так:

```
name1 = 'Maryna'
name2 = 'Gvido'
name1, name2 = name2, name1
```

Примітки

Примітка 1. Назви змінних нічого не говорять інтерпретатору, і навіть у дуже добре названій змінній не з'явиться потрібне значення, якщо ми його туди не запишемо.

Примітка 2. Нове значення змінної витісняє старе. Важливо уявляти, чому дорівнює значення змінної в кожен момент часу.

Примітка 3. Змінні можна вводити будь-якої миті (не тільки на початку програми).

Примітка 4. В якості назв змінних заборонено використовувати ключові (зарезервовані) слова. До ключових слів у мові Python відносяться:

Keywords in Python programming language				
False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Список ключових слів можна отримати підключивши модуль keyword і скориставшись командою

```
>>> import keyword
>>> print("Python keywords: ", keyword.kwlist)
```

PEP 8

При оформленні програм ми будемо користуватися ***PEP 8 - Python Enhanced Proposal.*** Цей документ пропонує єдиний та загальноприйнятий стиль написання програм мовою Python. Документ створено за рекомендаціями Гвідо Ван Россума, автора Python.

Деякі рекомендації PEP 8

Рекомендація 1. Уникайте використання пробілів перед дужкою, після якої починається список аргументів функції.

Правильно: `print('Follow PEP8!')`
Неправильно: `print ('Follow PEP8!')`

Рекомендація 2. Після коми потрібен пробіл.

Правильно: `print('PEP8', 'Rocks!')`
Неправильно: `print('PEP8','Rocks!')`

Рекомендація 3. Не відокремлюйте пробілами знак «рівно», коли він використовується для позначення значення параметра за промовчанням.

Правильно: `print('My name', 'is', 'Python', sep='**', end='+')`
Неправильно: `print('My name', 'is', 'Python', sep = '**', end = '+')`

PEP 8 у PyCharm

Середовище PyCharm має вбудовану підтримку форматування коду за стандартом PEP 8. Команда `Alt + Ctrl + l` виправить проблеми із форматуванням, але повністю до pep8 код не призведе.

Коментарі. До цього моменту все, що ми набирали в тексті наших програм, були команди для комп'ютера. Але в програму є сенс включати також примітки, що описують, що вона робить і як працює. Це може допомогти вам чи комусь іншому зрозуміти принцип роботи програми через деякий час.
Такі примітки називаються ***коментарями***.

Однорядкові коментарі

Будь-який рядок можна перетворити на коментар, помістивши перед ним символ #.
Розглянемо наступний код:

```
# Це коментар у програмі Python.
print('Python rocks!')
```

Якщо запустити цю програму, то вийде:

```
Python rocks!
```

Перший рядок під час запуску ігнорується. Коментар, що починається з символу #, призначений тільки для автора програми та для тих, хто читатиме цей код.

Коментар наприкінці рядка

Коментар можна розмістити після рядка коду.

```
print('Python rocks!') # Це коментар у програмі мовою Python
```

Коментар починається після символу #. Все, що знаходиться до цього символу, є звичайним кодом. Якщо запустити цю програму, вийде:

```
Python rocks!
```

Примітки

Примітка 1. Відповідно до стандарту PEP 8, коментарі повинні відокремлюватися хоча б двома пробілами від коду. Вони повинні починатися з символу # та одного пробілу.

Правильно:

```
# Далі буде надруковано текст
print('Comments in Python')  # Друк тексту за допомогою команди print.
```

Неправильно:

```
#Далі буде надруковано текст
print('Comments in Python')#Друк тексту за допомогою команди print.
```

Примітка 2. Під час введення коментарів у середовищі PyCharm ви виявите, що вони виділяються кольором. Це зроблено для полегшення читання коду.

Робота з цілими числами. Частина 1

Цілі числа. Усі попередні програми, які ми писали, працювали з текстовими даними. Справді, команда `input()` зчитує рядок тексту. Однак у багатьох випадках нам потрібно працювати саме із числами. Щоб у Python створити змінну цілого типу даних, потрібно опустити лапки при оголошенні змінної. Розглянемо наступний код:

```
num1 = 7 # num1 - це число
num2 = 10 # num2 - це число
num3 = num1 + num2 # num3 - це число
print(num3)
```

Внаслідок виконання такої програми буде виведено число 17.

Запам'ятай: числа позначаються без лапок, а рядки - з лапками.

Основні операції з числами

У мові Python, як і в математицы, над числами можна здійснювати 4 основні операції (+, −, *, /).

Оператор	Опис
+	додавання
−	віднімання
*	множення
/	поділ

Розглянемо таку програму:

```
a = 3
b = 2
print(a + b)
print(a - b)
print(a * b)
print(a / b)
```

Результатом виконання такої програми будуть числа:

```
5
1
6
1.5
```

Порядок виконання операцій

У математиці існує порядок виконання операцій, що визначає, які операції повинні виконуватися раніше за інших, навіть якщо у виразі вони написані правіше. Порядок виконання операцій у Python аналогічний порядку виконання операцій, які ви вивчали під час занять математики.

Розглянемо наступний код:

```
num1 = 2 + 3*4
num2 = (2 + 3) * 4
print(num1)
print(num2)
```

Результатом виконання такої програми будуть числа 14 і 20. У змінній `num1` зберігається число 14, оскільки насамперед виконується множення, а вже потім додавання. У змінній `num2` зберігається число 20, оскільки дужки змінили пріоритет виконання операцій.

Запам'ятай: насамперед виконується множення або розподіл, потім додавання і віднімання. Для зміни порядку виконання операцій знадобляться дужки.

Перетворення типів

Перетворення рядка до цілого числа. Для того, щоб перетворити рядок до цілого числа, ми використовуємо команду `int()`. Розглянемо наступний код:

```
s = '1992'
year = int(s)
```

Змінна `s` має рядковий тип даних. За допомогою команди `int()` ми перетворили рядок до цілого числа і записали результат в змінну з ім'ям `year`. ***Ми використовуємо нову команду (точніше, функцію)*** – `int()`. Вона означає таке: «Візьми те, що зазначено як аргумент у дужках, і перетвори це на ціле число». Таким чином, змінна `year` має цілий тип даних, а змінна `s` – рядковий тип даних.

Напишемо програму, яка зчитує два цілих числа і виводить на екран їхню суму. Наступний код тут не спрацює:

```
num1 = input()
num2 = input()
print(num1 + num2)
```

Python вважає, що в змінних `num1` та `num2` знаходиться текст, оскільки команда `input()` за умовчанням зчитує саме текст. Для того, щоб явно вказати, що потрібно працювати зі змінними цілого типу, треба написати так:

```
num1 = int(input())
num2 = int(input())
print(num1 + num2)
```

Запам'ятай: для того, щоб зчитати одне ціле число, ми пишемо наступний код:

```
num = int(input())
```

Перетворення цілого числа на рядок. Щоб перетворити ціле число в рядок, ми використовуємо команду `str()`. Розглянемо наступний код:

```
num = 17
s = str(17)
```

Змінна `num` має цілий тип даних. За допомогою команди `str()` ми перетворили ціле число в рядок і записали результат в змінну `s`. Таким чином, змінна `num` має цілий тип даних, а змінна `s` рядковий тип даних.

Примітки

Примітка 1. При роботі з цілими змінними часто потрібно здійснювати присвоєння значенню змінної цілої математичної формули:

<ім'я змінної> = <математична формула>

Примітка 2. Скорочення `int` походить від анлійського ***integer*** – цілий.
Примітка 3. У більшості мов програмування змінна цілого типу має обмеження. Наприклад, у мові C# змінна цілого типу може приймати значення діапазону $[-2^{63};2^{63-1}]$. У Python реалізована так звана ***довга арифметика***, тобто по суті змінна цілого типу не має обмежень.
Примітка 4. Мінус може бути використаний для позначення негативних чисел. А ось операції +, / і * завжди стоять між двома змінними та щось із ними роблять.

```
num1 = -6 # унарний мінус
num2 = 17 - 7 # бінарний мінус
```

Виконання задач 12-20 з практичної роботи 1

Робота з цілими числами. Частина 2

Додаткові операції. Ми познайомилися з 4 основними математичними операціями у мові Python: сума (+), різниця (−), добуток (*) та поділ (/). Додавши ще три операції, ми отримаємо інструментарій, достатній для написання 99% програм.

Оператор	Опис
**	Зведення в ступінь
%	Залишок від ділення
//	Цілочисельний поділ

Зведення в ступінь

Оператор зведення ступінь `a ** n` зводить число α в ступінь n.

Розглянемо роботу даного оператора на прикладах:

```
print(2 ** 0)
print(2 ** 1)
print(2 ** 2)
print(2 ** 3)
print(2 ** (-1))
```

Результатом виконання такої програми буде:

```
1
2
4
8
0.5
```

Зверніть увагу: оператор зведення у ступінь ** може зводити не лише у позитивний ступінь, а й у негативний. Аналогічно, підстава ступеня α також може бути числом негативним.
Запам'ятай: на відміну від багатьох мов програмування, у мові Python для зведення у ступінь використовується оператор **, а не ^.

Цілочисельний поділ

Для позитивних чисел оператор целочисленного поділу веде себя як звичайний поділ, крім того, що він відкидає десяткову частину результату. Розглянемо роботу даного оператора на прикладах:

```
print(10 // 3)
print(10 // 4)
print(10 // 5)
print(10 // 6)
print(10 // 12)
```

Результатом виконання такої програми буде:

```
3
2
2
1
0
```

При розподілі негативних чисел необхідно пам'ятати, що результат цілого розподілу не перебільшує частку. Іншими словами, округлення береться в меншу сторону (число -4 менше, ніж число -3).

Результатом роботи наступної програми:

```
print(10 // 3)
print(-10 // 3)
```

будуть числа:

```
3 # округлення у менший бік
-4 # округлення в меншу сторону
```

Поділ із залишком

Оператор поділу із залишком повертає залишок від поділу двох цілих чисел. Розглянемо роботу даного оператора на прикладах:

```
print(10 % 3)
print(10 % 4)
```

```
print(10 % 5)
print(10 % 6)
print(10 % 12)
```

Результатом виконання такої програми буде:

1
2
0
4
10

Запам'ятай: при знаходженні залишку від поділу на ціле число *n* може отримуватися результат 0,1,2,…, *n* −1. Наприклад, при розподілі на 2 можливі залишки 0,1, при розподілі на 3 можливі залишки 0,1,2 і т.д.

Примітки

Примітка 1. Оператор знаходження залишку дуже корисний під час вирішення багатьох задач. Наприклад, число ділиться на *n* націло тоді і лише тоді, коли залишок від поділу на *n* дорівнює 0.

Примітка 2. Оператори // та % мають такий самий пріоритет, як і оператори множення та звичайного поділу.

Примітка 3. Найвищий пріоритет має оператор зведення на ступінь **.

Примітка 4. Корисно прочитати про модульну арифметику в математиці. https://uk.wikipedia.org/wiki/Модульна_арифметика

Виконання задач 21-26 з практичної роботи 1

Обробка цифр числа

За допомогою операції знаходження залишку і цілісного поділу можна досить легко обчислити будь-яку цифру числа.

Розглянемо програму отримання цифр ***двозначного числа***:

```
num = 17
a = num % 10
b = num // 10
print(a)
print(b)
```

Результатом виконання програми будуть два числа:

7
1

Тобто, спочатку ми вивели останню цифру числа, а потім першу цифру.

Запам'ятай: остання цифра числа завжди визначається як ***залишок від поділу*** числа на `10 (% 10)`. Щоб відщепити останню цифру від числа, необхідно розділити його ***націло*** на `10 (// 10)`.
Розглянемо програму отримання цифр тризначного числа:

```
num = 754
a = num % 10
b = (num % 100) // 10
c = num // 100
print(a)
print(b)
print(c)
```

Результатом виконання програми будуть три числа:

```
4
5
7
```

Тобто, спочатку ми вивели останню цифру числа, потім середню цифру, а потім першу цифру.

Алгоритм отримання цифр *n*-значного числа

Нескладно зрозуміти, яким алгоритмом можна знайти кожну цифру *n*-значного числа *num*:

Остання цифра: $(num \% 10^1) // 10^0$;
Передостання цифра: $(num \% 10^2) // 10^1$;
Передпередостання цифра: $(num \% 10^3) // 10^2$;
…
Друга цифра: $(num \% 10^{n-1}) // 10^{n-2}$;
Перша цифра: $(num \% 10^n) // 10^{n-1}$.

Задача 1. Напишіть програму, яка визначає число десятків та одиниць у двозначному числі.

Рішення. Число одиниць – це остання цифра числа, число десятків – перша цифра. Щоб отримати останню цифру будь-якого числа, потрібно знайти залишок від розподілу числа на 10. Щоб знайти першу цифру двозначного числа, потрібно поділити число націло на 10. Програма, що вирішує поставлену задачу, може мати такий вигляд:

```
num = int(input())
last_digit = num % 10
first_digit = num // 10
```

```
print('Кількість десятків =', first_digit)
print('Кількість одиниць=', last_digit)
```

Результат:

```
59
Число десятків = 5
Число одиниць = 9
```

Задача 2. Напишіть програму, де розраховується сума цифр двозначного числа.

Рішення. Програма, що вирішує поставлену задачу, може мати такий вигляд:

```
num = int(input())
last_digit = num % 10
first_digit = num // 10
print('Сума цифр =', last_digit + first_digit)
```

Результат:

```
59
Сума цифр = 14
```

Задача 3. Напишіть програму, яка друкує число, утворене при перестановці цифр двоцифрового числа.

Рішення. Програма, що вирішує поставлену задачу, може мати такий вигляд:

```
num = int(input())
last_digit = num % 10
first_digit = num // 10
print("Шукане число =", last_digit * 10 + first_digit)
```

Результат:

```
59
Шукане число = 95
```

Задача 4. Напишіть програму, в яку вводиться тризначне число і яка виводить на екран його цифри (через кому).

Рішення. Програма, що вирішує поставлену задачу, може мати такий вигляд:

```
num = int(input())
digit3 = num % 10
digit2 = (num // 10) % 10
```

```
digit1 = num // 100
print(digit1, digit2, digit3, sep=',')
```

Результат:

```
596
5,9,6
```

Виконання задач 27-29 з практичної роботи 1

Лекція 2. Оператор розгалуження. Типи даних.

Анотація. Лекція присвячена умовному оператору if-else. Вивчимо спосіб роботи логічних операторів у Python та пріоритетність їх виконання. Вивчимо вкладений та каскадний умовний оператор.

План:
1. Умовний оператор
2. Відступи
3. Оператори порівняння
4. Розв'язання задач
5. Логічне множення `and`
6. Логічне додавання `or`
7. Логічне заперечення `not`
8. Розв'язання задач
9. Вкладені умови
10. Каскадні умови
11. Розв'язання задач

Умовний оператор if-else

Ми познайомилися з базовими будівельними блоками програм, навчилися писати програми, які забезпечують введення, обробку та виведення даних. Більше того, вміємо працювати з рядками та числами, як ми робимо це в математиці. Тепер познайомимося із керуванням ходом виконання програми.

Програми повинні вміти виконувати різні дії залежно від введених даних. Для прийняття рішення програма перевіряє, істинна чи хибна певна умова.

У Python існує кілька способів перевірки, і в кожному випадку можливі два результати: істина (True) або хибність (False).

Перевірка умов та прийняття рішень за результатами цієї перевірки називається ***розгалуженням*** (branching). Програма в такий спосіб вибирає, якою з можливих гілок їй рухатися далі.

У Python перевірка умови здійснюється за допомогою ключового слова `if`.

Розглянемо таку програму:

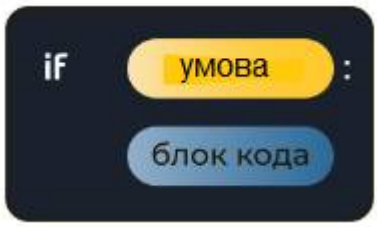
```
answer = input('Яку мову програмування ми вивчаємо?')
if answer == 'Python':
     print('Вірно! Ми ботаємо Python =)')
     print('Python - чудова мова!')
```

Програма просить користувача ввести текст та перевіряє результат введення. Якщо введений текст дорівнює рядку «Python», то виводить користувачеві текст:

```
Правильно! Ми ботаємо Python =)
Python - чудова мова!
```

Двокрапка (:) в кінці рядка з інструкцією if повідомляє інтерпретатору Python, що далі знаходиться ***блок команд***. До блоку команд входять усі рядки з відступом під рядком з інструкцією `if`, аж до наступного рядка без відступу.
Якщо умова є істинною, виконується весь розташований нижче блок. У попередньому прикладі блок інструкцій складає третій і четвертий рядки програми.

Блоком коду називають об'єднані один з одним рядки. Вони завжди пов'язані з певною частиною програми (наприклад, з інструкцією `if`). У Python блоки коду формуються за допомогою ***відступів***.



Попередня програма виводить текст у випадку, якщо умова є істинною. Але якщо умова є хибною, то програма нічого не виводить. Для того, щоб забезпечити можливість виконувати будь-що, якщо умова виявилася хибною, ми використовуємо ключове слово `else`.

```
answer = input('Яку мову програмування ми вивчаємо?')
if answer == 'Python':
     print('Вірно! Ми ботаємо Python =)')
     print('Python - чудова мова!')
else:
     print('Не зовсім так!')
```

У новій програмі ми обробляємо відразу два випадки: якщо умова є істинною (користувач ввів «Python»), і якщо умова є хибною (користувач ввів будь-що, крім «Python»).

Відступи

У деяких мовах програмування відступи — справа особистого смаку, і взагалі можна обходитися без них. Однак у Python вони є невід'ємною частиною коду. Саме відступ повідомляє інтерпретатору Python, де починається і закінчується блок коду.

Відступ - невелике зміщення рядка коду праворуч. На початку такого рядка знаходяться пробіли, і тому він на кілька символів відстоїть від лівого краю.

Деяким інструкціям в Python (наприклад, інструкції `if`) саме блок коду повідомляє, які дії слід зробити. Після `if` блок коду інформує інтерпретатор Python, як діяти, якщо умова істинна, і як — якщо вона хибна.

За узгодою PEP 8, для відступу блоків коду використовуються ***4 пробіли***. В середовищі PyCharm автоматично вставляються 4 пробіли.

Оператори порівняння

Можна помітити, що у перевірці умови ми використовували подвійну рівність (==), замість очікуваного одиночного (=). Не слід плутати ***оператор присвоєння*** (=) з ***умовним оператором*** (==).

Оператор присвоювання (=) присвоює змінним значення:

```
num = 1992
s = 'I love Python'
```

Для перевірки двох елементів на рівність Python використовує подвоєний знак (==). Ось так:

```
if answer == 'Python':

if name == 'Gvido':

if temperature == 40:
```

Плутанина з операторами == і = є однією з найпоширеніших помилок у програмуванні. Ці символи використовуються не тільки в Python, і кожен день багато програмістів використовують їх неправильно.

У Python існує 6 основних операторів порівняння.

Вираз	Опис
`if x > 7`	якщо x більше 7
`if x < 7`	якщо x менше 7
`if x >= 7`	якщо x більше або дорівнює 7
`if x <= 7`	якщо x менше або дорівнює 7
`if x == 7`	якщо x дорівнює 7
`if x != 7`	якщо x не дорівнює 7

Розглянемо приклад:

```
num1 = int(input())
num2 = int(input())

if num1 < num2:
     print(num1, 'менше ніж', num2)
if num1 > num2:
     print(num1, 'більше ніж', num2)
if num1 == num2: # використовуємо подвійну рівність
     print(num1, 'рівно', num2)
if num1! = num2:
     print(num1, 'не дорівнює', num2)
```

Ланцюжки порівнянь

Оператори порівняння в Python можна об'єднувати в ланцюжки (на відміну від більшості інших мов програмування, де для цього потрібно використовувати логічні зв'язки), наприклад, `a == b == c` або `1 <= x <= 10`. Наступний код перевіряє, чи знаходиться значення змінної `age` в діапазоні від 3 до 6:

```
age = int(input())
if 3 <= age <= 6:
     print('Ви дитина')
```

Код, що перевіряє рівність трьох змінних, може виглядати так:

```
if a == b == c:
     print('числа рівні')
else:
     print('числа не рівні')
```

Транзитивність

Операція рівності є ***транзитивною***.

Це означає, що якщо `a == b` та `b == c`, то з цього випливає, що `a == c`. Саме тому попередній код, що перевіряє рівність трьох змінних, працює як належить.

З курсу математики вам можуть бути знайомі інші приклади транзитивних операцій:

• ***Відношення порядку***: якщо a>b та b>c, то a>c;
• ***Паралельність прямих***: якщо a ∥ b і *b* ∥ *c*, то *a* ∥ *c*;
• ***Дільмість***: якщо *a* ділиться на *b* і *b* ділиться на *c*, *a* ділиться на *c*.

Наочно відносини порядку можна зрозуміти на такому прикладі: якщо сусід зліва старший за вас (a>b), а ви старші за сусіда справа (b>c), то сусід зліва точно старший за сусіда справа (a>c).

Операція нерівності (!=), на відміну операції рівності (==), є ***нетранзитивною***. Тобто з того, що a! = b і b! = c, зовсім не випливає, що a! = c. Справді, якщо вас звуть не так, як сусіда ліворуч і не так, як сусіда справа, то немає гарантії, що в обох сусідів не будуть однакові імена.

Таким чином, наступний код зовсім не перевіряє той факт, що всі три змінні різні:

```
if a != b != c:
     print('числа не рівні')
else:
     print('числа рівні')
```

Код, який перевіряє, що значення трьох змінних різні, ми навчимося писати трохи пізніше.

Розв'язання задач

Задача 1. Напишіть програму, яка зчитує один рядок. Якщо це рядок "Python", програма виводить "ТАК", інакше програма виводить "НІ".

Рішення. Програма, що вирішує поставлену задачу, може мати вигляд:

```
word = input()
if word == 'Python':
     print('ТАК')
else:
     print('НІ')
```

Задача 2. Напишіть програму, яка визначає, чи складається двозначне число, яке введене з клавіатури, з однакових цифр. Якщо складається, то програма виводить «ТАК», інакше програма виводить «НІ».

Рішення. Програма, що вирішує поставлену задачу, може мати вигляд:

```
num = int(input())
last_digit = num % 10 # остання цифра числа
first_digit = num // 10 # перша цифра числа

if last_digit == first_digit:
     print('ТАК')
else:
     print('НІ')
```

Задача 3. Напишіть програму, яка зчитує три числа та підраховує кількість парних чисел.

Рішення. Програма, що вирішує поставлену задачу, може мати вигляд:

```
num1, num2, num3 = int(input()), int(input()), int(input())
counter = 0 # змінна лічильник
if num1 % 2 == 0:
     counter = counter + 1 # збільшуємо лічильник на 1
if num2 % 2 == 0:
     counter = counter + 1 # збільшуємо лічильник на 1
if num3 % 2 == 0:
     counter = counter + 1 # збільшуємо лічильник на 1
print(counter)
```

Виконання задач 1-9 з практичної роботи 2

Логічні оператори

Як бути в ситуації, коли ми маємо кілька умов? У Python є три логічні оператори, які дозволяють створювати складні умови:

- `and` - логічне множення;
- `or` - логічне додавання;
- `not` – логічне заперечення.

Оператор and

Припустимо, ми написали програму для студентів віком від двадцяти років, які навчаються принаймні на 2 курсі. Доступ до неї тим, хто молодший, треба заборонити. Наступний код вирішує поставлену задачу:

```
age = int(input('Скільки вам років?: '))
grade = int(input('На якому курсі ви вчитеся?: '))
if age >= 20 and grade >= 2:
     print('Доступ дозволений.')
else:
     print('Доступ заборонено')
```

Ми об'єднали дві умови за допомогою оператора `and`. Це означає, що в цьому розгалуженні блок коду виконується лише при виконанні ***обох умов одночасно***!

Оператор `and` може об'єднувати довільну кількість умов:

```
age = int(input('Скільки вам років?: '))
grade = int(input('На якому курсі ви вчитеся?: '))
city = input('У якому місті ви живете?: ')
if age >= 20 and grade >= 2 and city == 'Одеса':
     print('Доступ дозволений.')
else:
     print('Доступ заборонено')
```

Це таблиця істинності оператора `and`. У ній перераховані вирази, з'єднані оператором `and`, показані всі можливі комбінації істинності та хибності та наведені результуючі значення виразів.

a	**b**	**a and b**
False	False	False
False	True	False
True	False	False
True	True	True

Як показує таблиця, щоб значення виразу з оператором `and` було істинним, мають бути істинними ***обидві (усі)*** об'єднані їм умови.

Оператор or

Оператор `or` також застосовується для об'єднання умов. Однак, на відміну від `and`, для виконання блоку коду достатньо виконання ***хоча б однієї з умов***.

```
city = input('У якому місті ви живете?: ')
if city == 'Одеса' or city == 'Київ' or city == 'Миколаїв':
     print('Доступ дозволений.')
else:
     print('Доступ заборонено')
```

Доступ буде дозволено, якщо хоча б одна з умов виконається.
Це таблиця істинності оператора `or`. У ній перераховані вирази, з'єднані оператором `or`, показані всі можливі комбінації істинності та хибності та наведені результуючі значення виразів.

a	b	a or b
False	False	False
False	True	True
True	False	True
True	True	True

Для того, щоб вираз `or` був істинним, потрібно, щоб ***хоча б одна*** умова оператора `or` була істинною. При цьому не має значення, істинним чи хибним є другий вираз.

Логічний вираз `X and Y` істинний, якщо ***обидва*** значення `X` і `Y` істинні.

Логічний вираз `X or Y` істинний, якщо хоча б одне із значень `X` і `Y` істинно.

Ми можемо використовувати обидва логічні оператори одночасно:

```
age = int(input('Скільки вам років?: '))
grade = int(input(''На якому курсі ви вчитеся?: '))
city = input('У якому місті ви живете?: ')
if age >= 20 and grade >= 2 and (city == 'Одеса' or city == 'Київ'):
     print('Доступ дозволений.')
else:
     print('Доступ заборонено')
```

Такий код перевіряє, що вік студентів від двадцяти років і навчаються вони принаймні на 2 курсі та живуть в Одесі чи Києві.

Оператор not

Оператор `not` дозволяє інвертувати (тобто замінити на протилежний) результат логічного вираження. Наприклад, наступний код:

```
age = int(input('Скільки вам років?: '))
if not (age < 20):
     print('Доступ дозволений.')
else:
     print('Доступ заборонено')
```

повністю еквівалентний коду:

```
age = int(input('Скільки вам років?: '))
if age >= 20:
     print('Доступ дозволений.')
else:
     print('Доступ заборонено')
```

У першому прикладі ми помістили вираз `age < 12` в дужки для того, щоб було чітко видно, що ми застосовуємо оператор `not` до значення виразу `age < 12`, а не тільки до змінної `age`.

Таблиця істинності для оператора `not`:

a	not a
False	True
True	False

Пріоритети логічних операторів

Логічні оператори, подібно до арифметичних операторів (+, -, *, /), мають пріоритет виконання. Пріоритет виконання наступний:

• в першу чергу виконується логічне заперечення `not`;
• далі виконується логічне множення `and`;
• далі виконується логічне додавання `or`.

Для ***явної вказівки порядку*** виконання умовних операторів ***використовують дужки***.

Примітки

Примітка 1. Частою помилкою у програмістів-початківців є плутанина логічних операторів `and` і `or`. Розглянемо дві умови:

```
if x > 1 and x < 100:
if x > 1 or x < 100:
```

Вірною є лише перша умова. У ній перевіряється, що число x знаходиться в діапазоні від 1 до 100, іншими словами, $x \in (1; 100)$. Друга умова перевіряє, що число x або більше 1, або менше 100. Однак такій умові задовольняє будь-яке число!

Примітка 2. Іншу часту помилку бачимо в наступному прикладі коду:

```
if age >= 7 and <= 9:
```

Запуск такого коду призведе до помилки під час виконання програми. Необхідно явно записувати умови:

```
if age >= 7 and age <= 9:
```

Примітка 3. Не забувайте, що Python має зручний спосіб для перевірки приналежності діапазону. Наприклад, наступний код:

```
if age >= 7 and age <= 9:
```

повністю еквівалентний коду:

```
if 7 <= age <= 9:
```

Останній код кращий.

Примітка 4. Обидва оператори `and` та `or` обчислюються за ***укороченою схемою***.

Ось як це працює з оператором `and`. Якщо умова ліворуч від оператора `and` є хибною, то умова праворуч від нього не перевіряється, оскільки результат висловлювання буде гарантовано хибним і перевірка умови, що залишилася, — марна трата процесорного часу.

Наприклад, у такому виразі:

```
5 > 100 та 10 > 0
```

обчислюється лише вираз `5 > 100`. Він хибний (бо 5 не може бути більше 100). При операторі `and` обидва вирази повинні бути правдивими, щоб результат був `True`. Але в нас вже одне не правдиве, значить, результат і так буде `False`. Тому нам і не треба обчислювати другий вираз, він все одно не вплине на результат.

Аналогічно працює оператор `or`. Якщо умова ліворуч від оператора `or` істинна, то умова праворуч від нього не перевіряється. Справді, результат буде гарантовано істинним і перевірка умови, що залишилася, стане порожньою витратою процесорного часу.

Наприклад, у такому виразі:

```
10 > 0 або 5 > 100
```

обчислюється лише вираз 10 > 0. Він правдивий, отже результат теж правдивий, тобто нам достатньо одного правдивого висловлювання при операторі `or`.

Розв'язання задач

Задача 1. Напишіть програму, яка визначає, чи задане натуральне число є тризначним.

Рішення. Програма, що вирішує поставлену задачу, може мати такий вигляд:

```
num = int(input())
if 100 <= num <= 999: # num >= 100 and num <= 999
     print('Число є тризначним')
else:
     print('Число не є тризначним')
```

Задача 2. Напишіть програму, яка перевіряє, що всі три цифри натурального тризначного числа є різними.

Рішення. Програма, що вирішує поставлену задачу, може мати такий вигляд:

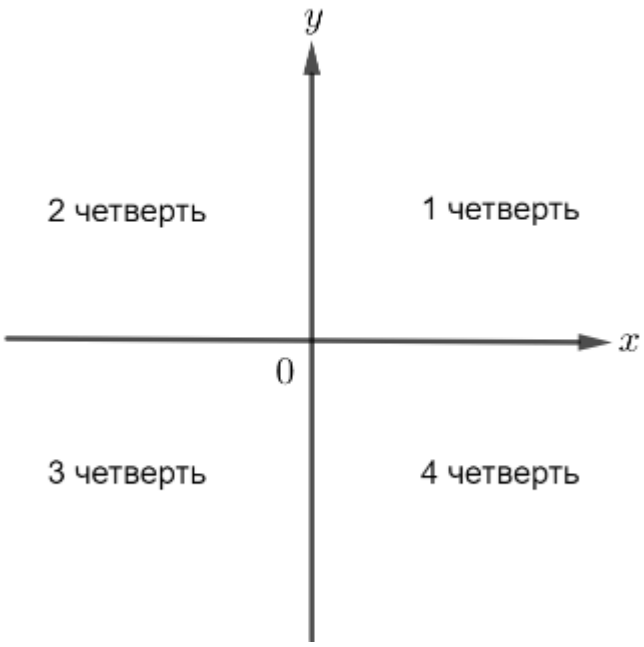
```
num = int(input())
d3 = num % 10
d2 = num % 100 // 10
d1 = num // 100
if d3 != d2 and d3 != d1 and d2 != d1:
     print('Цифри різні')
else:
     print('Цифри не різні')
```

Задача 3. Напишіть програму, яка за координатами точки, що не лежить на осях координат, визначає номер координатної чверті, де вона знаходиться.

Рішення. Програма, що вирішує поставлену задачу, може мати такий вигляд:

```
x = int(input())
y = int(input())

if x > 0 i y > 0:
     print('1 чверть')
if x < 0 i y > 0:
     print('2 чверть')
if x < 0 i y < 0:
     print('3 чверть')
if x > 0 i y < 0:
     print('4 чверть')
```



Виконання задач 10-17 з практичної роботи 2

Вкладений та каскадний умовний оператор.

Вкладений оператор

Всередині умовного оператора можна використовувати будь-які інструкції мови Python, у тому числі і умовний оператор. Отримуємо вкладене розгалуження: після однієї розвилки в ході виконання програми з'являється інша розвилка. При цьому вкладені блоки мають більший розмір відступу (+4 пробіли для кожного наступного рівня).

```
if умова1:
     блок коду
else:
     if умова2:
         блок коду
     else:
         if умова3:
             блок коду
         ...
```

У попередній темі ми розбирали задачу визначення координатної чверті точки. Програму можна переписати за допомогою вкладеного оператора:

```
x = int(input())
y = int(input())
if x > 0:
     if y > 0:
         print('Перша чверть')
     else:
```

```
        print('Четверта чверть')
else:
     if y > 0:
         print('Друга чверть')
     else:
         print('Третя чверть')
```

У цьому випадку рівень вкладеності дорівнює двом, отже програма однаково добре читається як з допомогою використання логічного оператора `and`, так і з допомогою вкладеного оператора.

Розглянемо програму, яка переводить стобальну оцінку до п'ятибальної. Для її реалізації необхідно користуватися вкладеним умовним оператором:

```
grade = int(input('Введіть вашу оцінку за 100-бальною системою: '))

if grade >= 90:
     print(5)
else:
     if grade >= 80:
         print(4)
     else:
         if grade >= 70:
             print(3)
         else:
             if grade >= 60:
                 print(2)
             else:
                 print(1)
```

У цьому прикладі рівень вкладеності настільки глибокий, що код важко зрозуміти.

Вибір із кількох альтернатив – це звичайна справа, тут є сенс уникати глибокого вкладення. Для цього в Python є ***каскадний умовний оператор***.

Ми не могли написати 5 незалежних `if`-ів, оскільки в такому разі було б надруковано одразу кілька значень п'ятибальної оцінки.

Каскадний умовний оператор

Якщо потрібно перевірити кілька умов, у мові Python використовується ***каскадний умовний оператор***.

Синтаксис каскадного умовного оператора має такий вигляд:

```
if умова1:
     блок коду
```

```
elif умова2:
     блок коду
...
else:
     блок коду
```

При виконанні такого умовного оператора спочатку перевіряється умова 1. Якщо вона є істинною, то виконується блок коду, який слідує відразу після неї, аж до виразу `elif`. Решта конструкції ігнорується. Однак якщо умова 1 є хибною, то програма перескакує безпосередньо до наступного виразу `elif` і перевіряє умову 2. Якщо вона є істинною, то виконується блок коду, який слідує відразу після неї, аж до наступного виразу `elif`. І решта умовного оператора тоді ігнорується. Цей процес триває доти, доки не буде знайдено умову, що є істинною, чи доки більше не залишиться виразів `elif`. Якщо жодна умова не є істинною, то виконується блок коду після вираження `else`.

Нижче наведений фрагмент коду є прикладом каскадного умовного оператора `if-elif-else`. Цей фрагмент коду працює так само, як попередній код, який використовує вкладений умовний оператор.

```
rade = int(input('Введіть вашу оцінку:'))

if grade >= 90:
     print(5)
elif grade >= 80:
     print(4)
elif grade >= 70:
     print(3)
elif grade >= 60:
     print(2)
else:
     print(1)
```

Зверніть увагу на вирівнювання та виділення відступом, які застосовані в інструкції `if-elif-else`: вирази `if, elif` та `else` вирівняні та виконувані за умовою блоки виділені відступом.

Інструкція `if-elif-else` не обов'язкова, тому що її логіка може бути запрограмована вкладеними інструкціями `if-else`. Однак довга серія вкладених інструкцій `if-else` має два характерні недоліки:
• програмний код може стати складним та важким для сприйняття;
• через необхідне виділення відступом тривала серія вкладених інструкцій `if-else` може стати занадто довгою, щоб повністю поміститися на екрані монітора без горизонтального прокручування.

Логіка інструкції `if-elif-else` зазвичай простежується легше, ніж довга серія вкладених інструкцій `if-else`. І оскільки в інструкції `if-elif-else` усі вирази вирівняні, довжина рядків у цій інструкції, як правило, коротша.

Запам'ятай. Заключний блок `else` в операторі `if-elif-else` є необов'язковим.

Розв’язання задач

Задача 1. Дано три цілих числа. Визначте, скільки з них збігаються. Програма повинна вивести одне з чисел: 3 (якщо всі збігаються), 2 (якщо два збігаються) або 0 (якщо всі числа є різними).

Рішення. Програма, що вирішує поставлену задачу, може мати такий вигляд:

1 спосіб. Використання вкладеного умовного оператора.

```
a, b, c = int(input()), int(input()), int(input())
if a == b:
     if b == c:
         print(3)
     else:
         print(2)
else:
     if a == c:
         print(2)
     else:
         if b == c:
             print(2)
         else:
             print(0)
```

2 спосіб. Використання каскадного умовного оператора.

```
a, b, c = int(input()), int(input()), int(input())
if a == b == c:
     print(3)
elif a == b != c:
     print(2)
elif a != b == c:
     print(2)
elif a == c != b:
     print(2)
else:
     print(0)
```

3 спосіб. Використання каскадного умовного оператора та логічного оператора `or`.

```
a, b, c = int(input()), int(input()), int(input())
if a == b == c:
     print(3)
elif a == b != c or a != b == c or a == c != b:
     print(2)
else:
     print(0)
```

Виконання задач 18-26 з практичної роботи 2

Лекція 3. Незмінювані типи даних. Модуль math

Анотація. Числові типи даних. Згадаймо особливості роботи з цілими числами, навчимося працювати з числами з плаваючою точкою. Вивчимо три вбудовані функції для роботи з числами `max, min, abs`. Поговоримо про рядковий тип даних та навчимося використовувати вбудовані функції len(), str(), а також працювати з операторами +, *, in. Модуль math, який містить математичні функції роботи з дійсними числами.

План:

1. Цілочисельний тип даних `int`
2. Числа з плаваючою точкою `float`
3. Вбудовані функції `max(), min(), abs()`
4. Розв'язання задач
5. Рядковий тип даних `str`
6. Функції `len()` та `str()`
7. Конкатенація рядків
8. Множення рядка на число
9. Оператор `in`
10. Розв'язання задач
11. Модуль `math`
12. Розв'язання задач

Числові типи даних

Цілочисельний тип даних

Цілі числа в Python представлені типом даних `int` (скорочення int походить від слова integer). Для визначення цілого числа типу `int` використається послідовність цифр від 0 до 9.

Очевидно вказане чисельне значення в коді програми називається ***цілочисельним літералом***. Коли Python зустрічає цілочисельний літерал, він створює об'єкт типу `int`, що зберігає вказане значення.

```
n = 17 # цілочисельний літерал
m = 7 # цілочисельний літерал
```

Цілочисленний тип даних `int` використовують не тільки тому, що він зустрічається в реальному світі, але й тому, що він природним чином виникає під час створення більшості програм.

Перетворення рядка на ціле число

Для перетворення рядка в ціле число ми використовуємо команду `int()`:

```
num = int(input()) # перетворення зчитаного рядка на ціле число
```

Для перетворення рядка на ціле число не обов'язково використовувати команду `input()`.

Наступний код перетворює рядок `12345` на ціле число:

```
n = int('12345') # перетворення рядка на ціле число
```

Якщо рядок не є числом, то при перетворенні виникне помилка.

Цілочисельні оператори

Мова Python надає чотири основні арифметичні оператори для роботи з цілими числами (+, −, *, /), а також три додаткові (% для залишку, // для цілочисельного поділу та ** для зведення в ступінь).

Наступна програма демонструє всі цілочисельні оператори:

```
a = 13
b = 7

total = a + b
diff = a - b
prod = a * b
div1 = a / b
div2 = a // b
mod = a % b
exp = a ** b

print(a, '+', b, '=', total)
print(a, '-', b, '=', diff)
print(a, '*', b, '=', prod)
print(a, '/', b, '=', div1)
print(a, '//', b, '=', div2)
print(a, '%', b, '=', mod)
print(a, '**', b, '=', exp)
```

В результаті роботи такої програми буде виведено:

```
13 + 7 = 20
13 - 7 = 6
13 * 7 = 91
13/7 = 1.8571428571428572
13 // 7 = 1
13% 7 = 6
13 ** 7 = 62748517
```

При звичайному розподілі (/) виходить число, що не є цілим. Поділ на нуль призводить до помилки.

Довга арифметика. Відмінною рисою мови Python є необмеженість цілочисельного типу даних. За фактом, розмір числа залежить від наявності вільної пам'яті на комп'ютері. Це відрізняє Python від таких мов як C++, C, C#, Java, де змінні цілого типу даних мають обмеження. Наприклад, у мові C# діапазон цілих чисел обмежений від -2^{63} до $2^{63}-1$.

```
atom = 10 ** 80 # кількість атомів у всесвіті
print('Кількість атомів =', atom)
```

Результатом виконання програми буде число із 81 цифрою:

```
Кількість атомів = 100000000000000000000000000000000000000000000000000000000000000000000000000000
```

Символ роздільник

Для зручного читання чисел можна використовувати символ підкреслення:

```
num1 = 25_000_000
num2 = 25000000
print(num1)
print(num2)
```

Результатом виконання такого коду буде:

```
25000000
25000000
```

Числа з плаваючою точкою

Нарівні з цілими числами в Python є можливість роботи з дробовими (дійсними) числами. Так, наприклад, числа $\frac{2}{3}$, $\sqrt{2}$, π – є дійсними і цілого типу `int` недостатньо для їхнього представлення.

Дробові (дійсні) числа в інформатиці називають ***числами з плаваючою точкою***.

Для представлення чисел з плаваючою точкою в Python використовується тип даних `float`.

```
e = 2.71828 # літерал з плаваючою точкою
pi = 3.1415 # літерал з плаваючою точкою
```

На відміну від математики, де роздільником є кома, в інформатиці використовується точка.

Перетворення рядка в число з плаваючою точкою

Для перетворення рядка до числа з плаваючою точкою ми використовуємо команду `float()`:

```
num = float(input()) # перетворення зчитаного рядка в число з плаваючою точкою
```

Для перетворення рядка до числа з плаваючою точкою необов'язково використовувати команду `input()`.

Наступний код перетворює рядок `1.2345` до числа з плаваючою точкою:

```
n = float('1.2345') # перетворення рядка до числа з плаваючою точкою
```

Якщо рядок не є числом, то при перетворенні виникне помилка.

Арифметичні оператори

Мова Python надає чотири основні арифметичні оператори для роботи з числами з плаваючою точкою (+, −, *, /) та один додатковий (** для зведення в ступінь).

Наступна програма демонструє арифметичні оператори:

```
a = 13.5
b = 2.0

total = a + b
diff = a - b
prod = a * b
div = a/b
exp = a ** b

print(a, '+', b, '=', total)
print(a, '-', b, '=', diff)
print(a, '*', b, '=', prod)
print(a, '/', b, '=', div)
print(a, '**', b, '=', exp)
```

В результаті роботи такої програми буде виведено:

```
13.5 + 2.0 = 15.5
```

```
13.5 – 2.0 = 11.5
13.5*2.0 = 27.0
13.5/2.0 = 6.75
13.5** 2.0 = 182.25
```

Поділ на нуль призводить до помилки.

Перетворення між int та float

Неявне перетворення. Будь-яке ціле число (тип `int`) можна використовувати там, де очікується число з плаваючою точкою (тип `float`), оскільки за необхідності Python автоматично перетворює цілі числа на числа з плаваючою точкою.

Явне перетворення. Число з плаваючою точкою не можна неявно перетворити на ціле число. Для такого перетворення необхідно використати явне перетворення за допомогою команди `int()`.

```
num1 = 17.89
num2 = -13.56
num3 = int(num1)
num4 = int(num2)
print(num3)
print(num4)
```

Результатом виконання такого коду буде:

```
17
-13
```

Зверніть увагу, що перетворення чисел з плаваючою точкою в ціле проводиться із заокругленням у бік нуля, тобто `int(1.7) = 1, int(-1.7) = -1`.

Не плутайте операцію перетворення та округлення. Для округлення чисел із плаваючою точкою використовуються додаткові команди. Про них розповімо пізніше.

Примітка

Початківців програмістів необхідність перетворення типів часто дратує, але досвідчені знають, що увага до типів даних — запорука успіху та спосіб уникнути помилок. В 1996 року французька ракета вибухнула в повітрі через проблему перетворення типів. Хоча помилка у вашій програмі може і не призвести до вибуху, все одно має сенс приділити час вивченню типів перетворення. Написавши кілька програм, ви переконаєтеся, що розуміння

типів даних допомагає не тільки писати компактний код, а й ясно викладати свої наміри, уникаючи помилок у нюансах.

Крах французької ракети «Аріан-5»

https://uk.wikipedia.org/wiki/Помилка_Аріан_5

Виконання задач 1-7 з практичної роботи 3

Вбудовані функції

Python включає багато заздалегідь визначених функцій. Програміст не бачить їхньої реалізації, вона прихована. Достатньо знати, як ці функції називаються і що вони роблять.

Ми вже стикалися із вбудованими функціями:

- `print()` - вивести на екран;
- `input()` - зчитати з клавіатури;
- `int()` – перетворити до цілого числа;
- `float()` — перетворити до числа з плаваючою точкою.

Розглянемо три нові вбудовані функції, які використовуються досить часто під час роботи з числами.

Функції min() та max()

Для визначення відповідно мінімального чи максимального значення використовуються функції `min()` та `max()`. Аргументів у цих функцій може бути будь-яка кількість, головне, щоб всі вони підтримували між собою операцію порівняння (наприклад, `float` і `int` підтримують порівняння, а `float` і `str` - ні).

Наприклад, результатом виконання наступного коду:

```
a = max(3, 8, -3, 12, 9)
b = min (3, 8, -3, 12, 9)
c = max(3.14, 2.17, 9.8)
print(a)
print(b)
print(c)
```

буде:

```
12
-3
9.8
```

Функція abs()

Модулем (абсолютною величиною) позитивного числа називається саме число, модулем негативного числа називається протилежне йому число, модуль нуля – нуль. Модуль числа a позначається $|a|$, для нього має місце рівність:

$$|a| = \begin{cases} a, & якщо \quad a > 0 \\ 0, & якщо \quad a = 0 \\ -a, & якщо \quad a < 0 \end{cases}$$

Для знаходження модуля (абсолютної величини) числа в Python використовується функція `abs()`.

Наприклад, результатом виконання наступного коду:

```
print(abs(10))
print(abs(-7))
print(abs(0))
print(abs(-17.67))
```

буде:

```
10
7
0
17.67
```

Зверніть увагу, всі три функції `max()`, `min()`, `abs()` працюють як з цілими числами, так і з числами з плаваючою точкою.

Виконання задач 8-12 з практичної роботи 3

Рядковий тип даних

Рядковий тип даних, як і числовий, часто використовується в програмуванні. В Python рядковий тип даних має назву `str` (скорочення від string - струна, ряд).

Для створення рядкової змінної (літералу), ми повинні укласти необхідний текст у лапки. У Python можна використовувати як одинарні лапки, так і подвійні:

```
s1 = 'Python rocks!'
s2 = "Python rocks!"
```

Нагадаємо, що за умовчанням команда `input()` зчитує саме рядок тексту:

```
s = input() # змінна s має рядковий тип str
```

Для завдання порожнього рядка ми використовуємо дві лапки однакового типу:

```
s1 = '' # порожній рядок
s2 = ' ' # рядок складається з одного символу пропуску
```

Не варто плутати порожній рядок і рядок, що складається з одного символу пропуску. Це абсолютно різні рядки.

Довжина рядка

Довжиною рядка називається кількість символів, з яких вона складається. Щоб порахувати довжину рядка, використовуємо вбудовану функцію `len()` (від слова length – довжина).

Наступний програмний код:

```
s1 = 'abcdef'
length1 = len(s1) # рахуємо довжину рядка із змінної s1
length2 = len('Python rocks!') # рахуємо довжину рядкового літералу
print(length1)
print(length2)
```

виведе:

```
6
13
```

При підрахунку довжини рядка рахуються всі символи, включаючи пробіли.

Перетворення чисел у рядок

Для перетворення рядка до числа ми використовували функції `int()` і `float()`. Для зворотного перетворення, тобто з числа в рядок, ми використовуємо функцію `str()`:

Розглянемо наступний програмний код:

```
num1 = 1777 # ціле число
num2 = 17.77 # число з плаваючою точкою
s1 = str(num1) # перетворили ціле число в рядок '1777'
s2 = str(num2) # перетворили число з плаваючою точкою в рядок '17.77'
```

Іноді працювати з рядками набагато простіше, ніж із числами. Навіть якщо в умові задачі сказано, що дається число, нам ніщо не заважає працювати з ним як із рядком.

Конкатенація рядків

Рядки, як і числа, можна складати. Операція складання рядків називається ***конкатенацією*** чи ***зчепленням***.

Розглянемо наступний програмний код:

```
s1 = 'ab' + 'bc'
s2 = 'bc' + 'ab'
s3 = s1 + s2 + '!!'
print(s1)
print(s2)
print(s3)
```

Результатом виконання такого коду буде:

```
abbc
bcab
abbcbcab!!
```

Операція складання рядків на відміну операції складання чисел не є комутативною, тобто, від перестановки місць доданків-рядків результат змінюється!

За допомогою конкатенації рядків можна емулювати виведення даних, яке раніше ми робили використовуючи необов'язкові параметри `sep` та `end`. Наступні два рядки коду роблять те саме:

```
print('a', 'b', 'c', sep='*', end='!')
print() # перехід на новий рядок
print('a' + '*' + 'b' + '*' + 'c' + '!')
```

Результатом виконання такого коду буде:

```
a*b*c!
a*b*c!
```

Множення рядка на число

У Python можна також множити рядок на число. Такий оператор повторює рядок вказану кількість разів.

Розглянемо наступний програмний код:

```
s = 'Hi' * 4
print(s)
```

Результатом виконання такого коду буде:

```
HiHiHiHi
```

Оператор множення рядка на число (repetition) дуже зручний на практиці. Наприклад, ми хочемо роздрукувати рядок, що складається з 75 символів `-`. Ми можемо це зробити за допомогою коду:

```
print('-' * 75)
```

Результатом виконання такого коду буде:

```
---------------------------------------------------------------------------
```

Рядок можна множити на число, але не можна множити на рядок.

Примітки

Примітка 1. Потрійні лапки в Python використовуються для багаторядкового тексту. Наприклад,

```
text = '''Python is interpreted, high-level, general-purpose programming language.
Created by Guido van Rossum and first released in 1991, Python design
```

```
philosophy emphasizes код readability with its notable use of significant whitespace.'''
```

Примітка 2. На перший погляд може здатися дивним, що можна використовувати як одинарні, так і подвійні лапки, однак такий підхід дозволяє легко додавати в рядок потрібні лапки:

```
s1 = 'Ми можемо використовувати в одиночних лапках подвійні лапки "Війна та мир"'
s2 = "Ми можемо використовувати в подвійних лапках одиночні лапки 'Війна і мир'"
print(s1)
print(s2)
```

Результатом виконання такого коду буде:

```
Ми можемо використовувати в одиночних лапках подвійні лапки "Війна та мир"
Ми можемо використовувати в подвійних лапках одиночні лапки 'Війна та мир'
```

Виконання задач 13-17 з практичної роботи 3

Оператор in

Python має спеціальний оператор `in`, який дозволяє перевірити, що один рядок знаходиться всередині іншого.

Розглянемо наступний код:

```
s = input()
if 'a' in s:
     print('Введений рядок містить символ a')
else:
     print('Введений рядок не містить символу a')
```

Такий код перевіряє, чи міститься в змінній `s` символ `a` і виводить відповідний текст.

Ми можемо використовувати оператор `in` разом із логічним оператором `not`. Наприклад

```
s = input()
if '.' not in s:
     print('Введений рядок не містить символу точки')
```

За допомогою оператора `in` ми можемо спростити наступний код, який перевіряє, що в змінній `s` знаходиться один із 5 символів `a, e, i, o, u`:

```
if s == 'a' or s == 'e' or s == 'i' or s == 'o' or s == 'u':
```

```
    print('YES')
```

до вигляду:

```
if len(s) == 1 and s in 'aeiou':
    print('YES')
```

Примітки

Примітка. Якщо рядок `s1` міститься у рядку `s2`, то кажуть, що рядок `s1` є підрядком для рядка `s2`. Іншими словами, оператор `in` визначає, чи є один рядок підрядком іншого.

Виконання задач 18-20 з практичної роботи 3

Модуль math

Модулі. Як уже говорилося, однією з переваг мови Python є безліч різноманітних функцій, які вже реалізовані та готові до використання. Такі функції упаковані у модулі. У Python модулем називається бібліотека функцій, яку можна підключати до своїх програм.

Модуль math

Модуль `math` – один з найважливіших у Python. Цей модуль надає великий функціонал щодо обчислень з дійсними числами (числами з плаваючою точкою).

Для використання цих функцій на початку програми необхідно підключити модуль, що робиться командою **import**:

```
import math

# програмний код
```

Після підключення модуля ми можемо використовувати його функції. Нехай ми хочемо:

- Обчислити $\sqrt{2}$ (корінь квадратний із двох);
- округлити число 3.8 до найближчого цілого числа вгору та вниз

Відповідний програмний код, який вирішує задачі виглядає так:

```
import math
```

```
num1 = math.sqrt(2) # обчислення квадратного кореня з двох
num2 = math.ceil(3.8) # округлення числа вгору
num3 = math.floor(3.8) # округлення числа вниз

print(num1)
print(num2)
print(num3)
```

і виводить:

```
1.4142135623730951
4
3
```

Особливості підключення модулів

Як можна помітити з прикладу вище, для виклику функції ми змушені вказувати назву модуля і символ точки. З іншого боку, якщо функції використовуються досить часто, такий виклик (постійна вказівка назви модуля і символу точки) може ускладнити програму і зробити її менш читабельною. Для того, щоб не вказувати назву модуля та символ точки під час виклику функцій, ми пишемо наступний код:

```
from math import *

num1 = sqrt(2) # обчислення кореня квадратного з двох
num2 = ceil(3.8) # округлення числа вгору
num3 = floor(3.8) # округлення числа вниз

print(num1)
print(num2)
print(num3)
```

Таким чином, підключення модуля наступним чином:

```
from math import *
```

дозволяє не писати назву модуля та символ точки. При такому способі підключення імпортуються **абсолютно** всі функції модуля `math`.

Якщо потрібно використовувати лише деякі функції модуля, то ми можемо імпортувати лише їх таким чином:

```
from math import sqrt, ceil
```

Тепер ми можемо викликати функції `sqrt()` і `ceil()` без префікса `math.`, однак ми не можемо викликати функцію `floor()`, оскільки вона не підключена:

```
from math import sqrt, ceil

print(sqrt(25))
print(ceil(34.7))

print(floor(12.8)) # призведе до помилки, оскільки функція floor не підключена
```

Список функцій модуля math

Список найчастіше використовуваних функцій модуля `math`:

Функція	**Опис**
Округлення	
int()	Округлює число у бік нуля
round(x)	Округлює число x до найближчого цілого. Якщо дробова частина дорівнює 0.5, то число округляється до найближчого парного числа
round(x, n)	Округлює число x до n знаків після точки
floor(x)	Округлює число x вниз («підлога»)
ceil(x)	Округлює число x вгору («стеля»)
abs(x)	Модуль числа x (абсолютна величина)
Корні, логарифми, ступеня та факторіал	
sqrt(x)	Квадратний корінь числа x
pow(x, n)	Зведення числа x у ступінь n
log(x)	Натуральний логарифм числа x. Основа натурального логарифму дорівнює числу e
log10(x)	Десятковий логарифм числа x. Основа десяткового логарифму дорівнює числу 10
log(x, b)	Логарифм числа x по основі b
factorial(n)	Факторіал натурального числа n
Тригонометрія	
degrees(x)	Перетворює кут x, заданий у радіанах, на градуси
radians(x)	Перетворює кут x, заданий у градусах, на радіани

cos(x)	Косинус кута x, що задається в радіанах
sin(x)	Синус кута x, що задається в радіанах
tan(x)	Тангенс кута x, що задається в радіанах
acos(x)	Повертає кут у радіанах від 0 до π, cos якого дорівнює x
asin(x)	Повертає кут у радіанах від $-\frac{\pi}{2}$ до $\frac{\pi}{2}$, sin якого дорівнює x
atan(x)	Повертає кут у радіанах від $-\frac{\pi}{2}$ до $\frac{\pi}{2}$, tan якого дорівнює x
atan2(y, x)	Полярний кут (у радіанах) точки з координатами (x, y)

Для отримання квадратного кореня можна скористатися кодом

```
n ** 0.5
```

замість `math.sqrt(n)`.

Список констант модуля math

Модуль `math` надає ряд вбудованих математичних констант:

Константа	Опис
pi	Число π=3.141592653589793
e	Число e = 2.718281828459045 (константа Ейлера)

Примітки

Примітка 1. Всі функції модуля `math` повертають значення, яке можна вивести на екран, привласнити іншій змінній або використовувати в математичному виразі.

Примітка 2. Для використання функцій `int()`, `float()`, `abs()`, `min()`, `max()`, `round()` підключати модуль `math` немає необхідності. Це так звані вбудовані функції.

Примітка 3. Виклик функцій `pow(x, n)` можна замінити за допомогою оператора зведення у ступінь: `x**n`.

Виконання задач 21-27 з практичної роботи 3

Лекція 4. Цикли for і while.

Анотація. Лекція присвячена циклу for. Що таке цикл for і як створювати програми, що повторюють певні дії? Поняття змінної циклу. Розглянемо функцію `range()`, яка дозволяє генерувати послідовність чисел. Вивчимо два додаткових параметра функції `range()`, які дозволяють налаштовувати елементи послідовності. Розглянемо часті сценарії під час написання циклів. Цикл while. Ми навчимося створювати програми, що повторюють певні дії, поки виконується деяка умова. Оператори break та continue, а також необов'язковий блок else. Проведення рев'ю коду, пошук помилок та аналіз продуктивності. Вкладені цикли, що знаходяться всередині інших циклів.

План:

1. Цикл for
2. Змінна циклу
3. Вирішення задач
4. Функція `range()` з одним параметром
5. Функція `range()` із двома параметрами
6. Функція `range()` із трьома параметрами
7. Вирішення задач
8. Підрахунок кількості
9. Обчислення суми та добутку
10. Обмін значень змінних
11. Сигнальні мітки
12. Визначення максимуму та мінімуму
13. Розширені оператори присвоєння (+=, -=, //= і т.д.)
14. Цикл while
15. Зчитування даних до стоп значення
16. Безкінечний цикл
17. Використання циклу while для обробки цифр числа
18. Вирішення задач
19. Рев'ю коду
20. Пошук помилок
21. Продуктивність коду
22. Вкладені цикли
23. Оператори break і continue у вкладених циклах
24. Вирішення задач

Цикл for

Одна з переваг комп'ютерів перед людьми - здатність повторювати ті самі дії багаторазово, швидко і зовсім не втомлюючись.

У Python існує два основних різновиди циклів:

- цикли, що повторюються певну кількість разів (for, лічильні цикли, ***counting loops***);
- цикли, що повторюються до настання певної події (while, умовні цикли, ***conditional loops***).

Цикл `for` чудово працює, якщо ми знаємо, скільки повторень (ітерацій) нам потрібно зробити.

Розглянемо код, який роздрукує 10 разів слово "Привіт":

```
for i in range(10):
     print('Привіт')
```

Структура циклу for в Python виглядає так:

```
for назва_змінної_циклу in range(кількість повторень):
     блок коду
```

Двокрапка (:) в кінці рядка з інструкцією `for` повідомляє інтерпретатору Python, що далі знаходиться ***блок команд***. До блоку команд входять усі рядки, розташовані з відступом від рядка з інструкцією `for`, аж до наступного рядка без відступу.

Блок команд, який виконується у циклі `for` називається ***тілом циклу***.

У попередніх темах ми зчитували кілька чисел, за допомогою кількох команд `input`. За допомогою циклу `for` можна зчитувати та обробляти скільки завгодно чисел.

Розглянемо наступний програмний код:

```
for i in range(5):
     num = int(input())
     print('Квадрат вашого числа дорівнює:', num * num)
print('Цикл завершений')
```

Така програма зчитує 5 чисел і виводить на екран їх квадрати. Оскільки другий та третій рядки виділені відступом, Python вважає, що це тіло циклу, яке виконується 5 разів. Четвертий рядок не містить відступу, тому не є частиною циклу і буде виконаний лише один раз, після того, як цикл завершиться.

Приклади використання циклу `for`

Розглянемо наступний програмний код:

```
print('A')
print('B')
for i in range(5):
     print('C')
     print('D')
print('E')
```

Результатом виконання такої програми будуть рядки

```
A
B
C
D
C
D
C
D
C
D
C
D
E
```

Тобто спочатку програма роздрукує символи `A` та `B`, потім символи `C` та `D` п'ять разів, а потім роздрукує символ `E` один раз. Тіло циклу складається з двох рядків: четвертого та п'ятого і саме вони повторюватимуться.

У програмі може бути скільки завгодно циклів. Наприклад, якщо ми хочемо, щоб спочатку 5 разів було роздруковано символ `C`, а потім 5 разів символ `D`, ми можемо використовувати 2 цикли:

```
print('A')
print('B')
for i in range(5):
     print('C')
for i in range(5):
     print('D')
print('E')
```

Результатом виконання такої програми будуть рядки:

```
A
B
C
C
C
C
C
```

```
D
D
D
D
D
E
```

Примітки

Примітка 1. Одноразове виконання тіла циклу називається ***ітерацією циклу***.

Примітка 2. Графічне представлення циклу має вигляд:

Примітка 3. Нагадаємо, що ***блоком коду*** називають об'єднані один з одним рядки. Вони завжди пов'язані з певною частиною програми (наприклад, з інструкцією `if` чи `for`). У Python блоки коду формуються за допомогою ***відступів***.

Примітка 4. Слово `for` пишеться маленькими літерами, перший рядок має закінчуватися двокрапкою, і тіло циклу має бути виділене відступом.

Вирішення задач 1-4 з практичної роботи 4

Змінна циклу

Погляньмо ще раз на базову структуру циклу `for`:

```
for назва_змінної_циклу in range(кількість повторень):
     блок коду
```

Не зовсім зрозуміло, навіщо потрібна і як працює змінна циклу.

Розглянемо наступний код:

```
for i in range(10):
     print(i)
```

Результатом виконання такого коду буде:

```
0
1
2
3
4
5
6
7
8
9
```

Коли цикл вперше починає роботу, Python встановлює значення змінної циклу `i = 0`. Щоразу, коли ми повторюємо тіло циклу, Python збільшує значення змінної на 1.

Чому більшість програмістів починають цикл із 0, а чи не з 1? Раніше деякі починали цикл з 1, а деякі з 0. Ті та інші наводили вельми витончені аргументи, сперечаючись про те, який спосіб краще. Але зрештою перемогли прихильники другого варіанта. З тих пір більшість починає цикли з 0. Зокрема, у Python цикл for починається з 0, однак у майбутньому ви дізнаєтесь, як це змінити.

Оскільки змінна циклу `i` збільшується на 1 щоразу, то її можна використовуватиме для відстеження номера ітерації, на якій ми перебуваємо у циклічному процесі.

Розглянемо наступний код:

```
for i in range(10):
     print(i, '- Привіт')
```

Результатом виконання такого коду буде:

```
0 - Привіт
1 - Привіт
2 - Привіт
3 - Привіт
4 - Привіт
5 - Привіт
6 - Привіт
7 - Привіт
8 - Привіт
9 – Привіт
```

Якщо ми хочемо почати з 1, то можемо написати код:

```
for i in range(10):
     print(i + 1, '- Привіт')
```

Результатом виконання такого коду буде:

```
1 - Привіт
2 - Привіт
3 - Привіт
4 - Привіт
5 - Привіт
6 - Привіт
7 - Привіт
8 - Привіт
9 - Привіт
10 – Привіт
```

Зверніть увагу, за рахунок виразу `i + 1` ми починаємо виведення з 1, а не з 0.

Імена змінних циклу

Раніше говорилося, що імена змінних повинні мати осмислений характер і описувати їхнє призначення. Однак для змінних циклу іноді робиться виняток. У програмуванні для змінних циклу зазвичай використовують літери `i, j, k`.

Наступні дві програми абсолютно однакові: у першій програмі змінна циклу має назву `i`, у другій програмі – `number`:

```
for i in range(10):
     print(i)

for number in range(10):
     print(number)
```

Результатом виконання обох програм буде:

```
0
1
2
3
4
5
6
7
8
9
```

Чому для змінної циклів зарезервовані літери `i, j, k`? Справа в тому, що раніше програми використовувалися для математичних розрахунків, а в математиці букви `a, b, c` та `x, y, z` вже зарезервовані для інших цілей.

Тому програмісти обрали для цієї мети змінні `i, j, k`, і це стало загальноприйнятою практикою.

Бувають ситуації, коли змінна циклу не використовується у тілі циклу. У такому разі замість того, щоб давати їй ім'я, ми можемо вказати символ нижнього підкреслення _:

```
for _ in range(5):
     print('Python - awesome!')
```

Результатом виконання такого коду буде:

```
Python - awesome!
Python - awesome!
Python - awesome!
Python - awesome!
Python - awesome!
```

Якщо змінна циклу не використовується в тілі циклу, вказуйте замість неї символ нижнього підкреслення _.

Примітка. Слід пам'ятати, що права границя циклу в Python завжди не включена. Таким чином наступний код:

```
for i in range(5):
     print(i)
```

роздрукує числа від 0 до 4:

```
0
1
2
3
4
```

Якщо потрібно роздрукувати числа від 1 до 5, ми пишемо код:

```
for i in range(5):
     print(i + 1)
```

Вирішення задач 5-8 з практичної роботи 4

Цикл for. Функції range()

Функція range() з одним параметром

Розглянемо програмний код:

```
for i in range(10):
     print('Привіт', i)
```

Значення, яке ми вказуємо в дужках, у функції `range()` позначає кількість ітерацій циклу, при цьому змінна `i` приймає послідовно значення: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Якщо бути більш точним, ми говоримо, що функція `range(n)` ***генерує послідовність*** чисел від 0 до *n*-1, а цикл `for` послідовно перебирає цю послідовність.

Перевантаження range() із двома параметрами

Якщо ми хочемо починати послідовність не з 0, а з якогось іншого числа, то ми можемо використовувати перевантаження функції `range()`, що приймає два параметри. Наприклад, виклик функції `range(1, 5)` згенерує послідовність чисел 1, 2, 3, 4 (будьте уважні, права межа не включена). Якщо нам потрібні числа від 1 до 5 включно, ми використовуємо `range(1, 6)`.

Таким чином:

- `range(n)`: створює послідовність чисел 0, 1, 2, 3, ..., n - 1;
- `range(n, m)`: створює послідовність чисел n, n + 1, n + 2, ..., m – 2, m – 1.

Напишемо програму, яка виводить ті числа з проміжку [100; 999], які закінчуються на 7.

Використовуючи функцію `range()` з двома параметрами, отримуємо:

```
for i in range(100, 1000): # перебираємо числа від 100 до 999
     if i % 10 == 7: # використовуємо залишок від розподілу на 10, для
отримання останньої цифри
         print(i)
```

Зверніть увагу, в якості другого параметру ми передали число 1000.

Якщо перший параметр більший за другий, то функція `range()` генерує порожню послідовність. Наприклад, виклик функції `range(10, 1)` призводить до створення порожньої послідовності.

Перевантаження range() із 3 параметрами

Передаючи два параметри у функцію `range()` ми можемо генерувати будь-яку послідовність цілих чисел з кроком 1. Але що робити якщо потрібно поміняти крок? Як бути, якщо ми хочемо згенерувати послідовність чисел 5, 10, 15, 20, 25? У цьому випадку існує ще одне навантаження функції `range()`, що приймає три параметри: `range(n, m, k)`. Перший параметр задає ***старт послідовності***, другий параметр задає ***стоп послідовності*** та третій – ***крок генерації*** чисел.

Наприклад, виклик функції `range(1, 10, 2)` створить послідовність чисел 1, 3, 5, 7, 9, а виклик функції `range(5, 30, 5)` згенерує послідовність 5, 10, 15, 20, 25.

Напишемо програму, яка виводить усі парні числа з проміжку [56; 170].

Використовуючи функцію `range()` з трьома параметрами, отримуємо:

```
for i in range(56, 171, 2):
     print(i)
```

Зверніть увагу, ми можемо використовувати функцію `range()` із двома параметрами:

```
for i in range(56, 171):
     if i % 2 == 0:
         print(i)
```

однак такий код виходить менш ефективним.

Від'ємний крок генерації

Якщо крок генерації є позитивним числом, то послідовність, що генерується, буде ***зростати***. Ми можемо вказати негативний крок генерації (третій параметр), що призведе до генерування ***спадної*** послідовності.

У разі від'ємного кроку, ми повинні гарантувати, що старт послідовності (перший параметр) більший за кінець послідовності (другий параметр).

Наприклад, виклик функції `range(20, 16, -1)` створить послідовність чисел 20, 19, 18, 17, а виклик функції `range(20, 10, -3)` згенерує послідовність 20, 17, 14, 11.

Напишемо програму, яка відраховує від 5 до 1, а потім виводить текст `Злітаємо!`

```
for i in range(5, 0, -1):
     print(i, end='')
print('Злітаємо!!!')
```

Результатом буде:

```
5 4 3 2 1 Злітаємо!!!
```

Якщо величина кроку від'ємна і перший параметр менший за другий, то функція `range()` генерує порожню послідовність. Наприклад, виклик функції `range(1, 10, -1)` призводить до генерації порожньої послідовності.

Приклади використання функції `range()`

Виклик функції	**Послідовність чисел**
`range(10)`	`0, 1, 2, 3, 4, 5, 6, 7, 8, 9`
`range(1, 10)`	`1, 2, 3, 4, 5, 6, 7, 8, 9`
`range(3, 7)`	`3, 4, 5, 6`
`range(7, 3)`	`порожня послідовність`
`range(2, 15, 3)`	`2, 5, 8, 11, 14`
`range(9, 2, -1)`	`9, 8, 7, 6, 5, 4, 3`
`range(3, 10, -2)`	`порожня послідовність`

Примітки

Примітка 1. Функція `range()` може приймати від одного до трьох параметрів: `range(n), range(n, m), range(n, m, k)`

- перший параметр – це старт послідовності (включно);
- другий параметр - це стоп послідовності (не включно);
- третій параметр – це величина кроку.

Примітка 2. Функція `range()` може генерувати лише цілі числа, включаючи від'ємні.

Примітка 3. Розмір кроку не може дорівнювати нулю. Наступний код:

```
for i in range(1, 10, 0):
     print(i)
```

приведе до помилки **ValueError: range() arg 3 must not be zero**.

Вирішення задач 9-13 з практичної роботи 4

Деякі сценарії використання циклів

Підрахунок кількості

Нерідко потрібно, щоб наші програми підраховували скільки разів щось трапилося. Наприклад відео гра може підраховувати кількість поворотів персонажа або математична програма може обчислювати скільки чисел мають деяку властивість. Ключ до підрахунку – використання ***змінної лічильника***.

Напишемо програму, яка зчитує 10 чисел і визначає скільки з них більше 10.

```
counter = 0
for _ in range(10):
     num = int(input())
     if num > 10:
         counter = counter + 1
print('Було введено', counter, 'чисел, більше ніж 10.')
```

Кожного разу коли ми читаємо число більше 10, ми додаємо 1 до нашого поточного значення змінної `counter`. У програмі це реалізовано в рядку `counter = counter + 1`. Зверніть увагу на початкове значення змінної лічильника `counter = 0`. Без початкового значення ми отримали б помилку, оскільки дійшовши до рядка `counter = counter + 1` Python нічого не знав про змінну `counter` . Рядок коду `counter = counter + 1` означає: візьми старе значення змінної `counter`, додай до нього 1 і перепривласній змінній це значення. Якщо не надати змінній початкове значення, то незрозуміло, до чого додавати 1 вперше.

Підрахунок кількості – це дуже частий сценарій. Він складається із двох кроків:

1. Створення ***змінної лічильника*** та надання їй первісного значення: `counter = 0`;
2. Збільшення змінної лічильника на 1: `counter = counter +1`.

Часто під час написання програм потрібно використовувати кілька лічильників. Модифікуємо попередню програму: порахуємо ще кількість нулів серед введених чисел.

```
counter1 = 0
counter2 = 0
for _ in range(10):
     num = int(input())
     if num > 10:
         counter1 = counter1 + 1
     if num == 0:
```

```
        counter2 = counter2 + 1
print('Було введено', counter1, 'чисел, більше ніж 10.')
print('Було введено', counter2, 'нулів.')
```

Розглянемо ще один приклад: підрахувати кількість чисел із діапазону [1;100], квадрат яких закінчується на 4.

```
counter = 0
for i in range(1, 101):
     if i**2 % 10 == 4:
         counter = counter + 1
print(counter)
```

Ми використовуємо функцію `range()` з двома параметрами для генерації послідовності чисел від 1 до 100. Змінна `i` послідовно набуває значень від 1 до 100, ми перевіряємо, умова: квадрат числа `i` закінчується на 4 за допомогою умови `i**2 % 10 == 4`.

Для змінної лічильника зручно використовувати ім'я `counter`.

Обчислення суми та добутку

Поруч із підрахунком кількості по частоті стоїть задача обчислення суми. Наприклад, відеогра повинна рахувати суму очок. У такому разі початкове значення змінної дорівнюватиме 0, а далі воно збільшуватиметься на деяку кількість зароблених очок, скажімо на 10. Ми пишемо наступний код:

```
score = 0
...
score = score + 10
```

Напишемо програму, яка зчитує 10 чисел та визначає суму тих із них, які більше 10.

```
total = 0
for _ in range(10):
     num = int(input())
     if num > 10:
         total = total + num
print('Сума чисел більших 10 дорівнює', total)
```

Щоразу, коли програма зчитує число більше 10, вона додає його до поточного значення змінної `total`. Це реалізовано у рядку `total = total + num`. Зверніть увагу на початкове значення змінної суматора `total = 0`. Без початкового значення ми отримали б помилку, оскільки дійшовши до рядка `total = total + num` Python нічого не знав би про змінну `total` . Рядок коду `total = total + num` означає візьми старе значення змінної `total`, додай до

нього `num` і перепривласній змінній це значення. Якщо не надати змінній початкове значення, то ні до чого додавати `num` вперше.

Підрахунок суми складається з двох кроків:

1. Створення змінної ***суматора*** та надання їй первісного значення: `total = 0`;
2. Збільшення змінної суматора на потрібне число: `total = total + num`.

Напишемо програму, яка розраховує суму натуральних чисел від 1 до 100:

```
total = 0
for i in range(1, 101):
     total = total + i
print('Сума дорівнює', total)
```

Розглянемо ще один приклад: напишемо програму, яка запитує 10 цілих чисел і знаходить їхнє середнє значення:

```
total = 0
for _ in range(10):
     num = int(input())
     total = total + num
average = total / 10
print('Середнє значення дорівнює', average)
```

Аналогічним чином обчислюється добуток. При обчисленні добутку початкове значення змінної ***мультиплікатора*** ми встановлюємо рівним 1, на відміну від суматора, де воно дорівнює 0.

Для змінної суматора та мультиплікатора зручно використовувати ім'я `total`.

Обмін значень змінних

Дуже часто нам потрібно обміняти значення двох змінних `x` та `y`. Початківці програмісти іноді пишуть такий код:

```
x = y
y = x
```

Однак він не працює. Припустимо, що `x = 3` і `y = 5`. Перший рядок надасть змінній `x` значення 5, що правильно, однак другий рядок встановить значення змінної `y` в 5, оскільки значення `x` вже дорівнює 5. Для вирішення задачі ми можемо використовувати ***тимчасову змінну***:

```
temp = x
x = y
y = temp
```

Такий код пишуть майже у всіх мовах програмування. Однак у Python є і простіший спосіб. Ми можемо написати так:

```
x, y = y, x
```

В результаті виконання такого коду Python змінює значення змінних `x` та `y` місцями.

Сигнальні мітки

Сигнальна мітка (прапорець) може використовуватися, коли треба щоб одна частина програми дізналася про те, що відбувається в іншій частині програми.
Напишемо програму, яка визначає, що натуральне число є ***простим***:
https://uk.wikipedia.org/wiki/Просте_число

```
num = int(input())
flag = True

for i in range(2, num):
     if num % i == 0: # якщо вихідне число ділиться на якесь відмінне
від 1 і самого себе
          flag = False

if num == 1:
     print('Це одиниця, вона не проста і не складова')
elif flag == True:
     print('Число просте')
else:
     print('Число складове')
```

Нагадаємо, що число є простим, якщо воно не має дільників, крім 1 і самого себе. Наведена вище програма працює наступним чином: початкове значення змінної прапора дорівнює `True`, що говорить про те, що число є простим. Потім ми перебираємо всі числа від 2 до `num - 1` (включно). Якщо одне з цих значень є дільником числа `num`, тоді число `num` є складовим і ми встановлюємо значення прапора `False`. Як тільки цикл завершено, ми перевіряємо, чи встановлено прапор чи ні. Якщо це так, ми знаємо, що був дільник і число не є простим. В іншому випадку число має бути простим.

Прапорні змінні можуть мати більш осмислену назву. Наприклад, у випадку з перевіркою числа на простоту назва прапорової змінної могла б бути `is_prime`.

Максимум та мінімум

Пошук найбільшого чи найменшого значення у певній послідовності чисел, також найчастіша задача у програмуванні. Напишемо програму, яка зчитує ***10 позитивних чисел*** і знаходить серед них найбільше число.

```
largest = -1
for _ in range(10):
     num = int(input())
     if num > largest:
         largest = num
print('Найбільше число дорівнює', largest)
```

Ми встановлюємо початкове значення змінної `largest` в -1. Далі програма зчитує 10 чисел, і якщо якесь з них виявляється більшим за поточне значення `largest`, перепривласнює його. В якості початкового значення взято число -1, оскільки знаємо, що всі числа позитивні, в такий спосіб вже перше порівняння призведе до переприсвоювання.

Поширений підхід, коли як початкове значення змінної, відразу приймає перший елемент послідовності. Напишемо програму, яка зчитує 10 чисел (необов'язково позитивних) і знаходить серед них найбільше:

```
largest = int(input()) # приймаємо перше число за максимальне
for _ in range(9):
     num = int(input())
     if num > largest:
         largest = num
print('Найбільше число дорівнює', largest)
```

Для знаходження найменшого значення послідовності слід поміняти знак нерівності (>) на протилежний (<). У такому випадку назву змінної `largest` варто замінити на `smallest`.

Для змінних, що зберігають найбільше та найменше значення, підходять імена `largest` та `smallest`.

Розширені оператори присвоєння

Досить часто програми мають інструкції присвоювання, в яких змінна на лівій стороні від оператора = також з'являється на правій стороні. Наприклад,

```
counter = counter + 1
```

На правій стороні оператора присвоєння 1 додається змінної `counter`. Отриманий результат присвоюється змінної `counter`, замінюючи початкове значення. Насправді, цей рядок коду додає 1 до `counter`. Ще один приклад такої інструкції ми бачили при підрахунку суми:

```
total = total + num
```

Ця інструкція надає значення виразу `total + num` змінної `total`. В результаті виконання цієї інструкції число `num` додається до значення `total`.

Різні інструкції присвоєння (у кожній інструкції `x = 6`)

Інструкція	Що вона робить	Значення x після інструкції
x = x + 4	Додає 4 до x	10
x = x – 3	Віднімає 3 з x	3
x = x * 10	Помножує x на 10	60
x = x / 4	Ділить x на 4	1.5
x = x // 4	Ділить націло x на 4	1
x = x % 4	Знаходить залишок від поділу x на 4	2

Ці типи операцій знаходять широке застосування у програмуванні. Для зручності Python пропонує ***розширені оператори присвоєння***. Розширені оператори не вимагають, щоб програміст двічі набирав ім'я змінної. Наведену нижче інструкцію:

```
total = total + num
```

можна переписати як

```
total += num
```

Так само інструкцію

```
counter = counter + 1
```

можна переписати як

```
counter += 1
```

Оператор	Приклад використання	Еквівалент
+=	x += 5	x = x + 5
-=	x -= 2	x = x - 2
*=	x *= 10	x = x * 10

/	= x / = 4	x = x / 4
//=	x //= 4	x = x // 4
%=	x %= 4	x = x % 4

Примітки

Примітка 1. Аналогічно можна змінювати місцями значення трьох і більше змінних.

```
a, b, c, d = b, c, d, a
```

Примітка 2. Дуже часто сигнальні позначки називають `flag`.

Примітка 3. Оскільки Python має вбудовані функції `max()` і `min()`, то давать такі назви для максимального та мінімального значення не дуже добре. Куди краще використовувати назви `largest` і `smallest` або `mx` і `mn`.

Примітка 4. Суму чисел від 1 до 100 можна обчислити і без циклу:

$$Сума = \frac{1+100}{2} \cdot 100 = 5050$$

Справді, числа від 1 до 100, можна розбити на 50 пар, сума в яких дорівнює 101:

1 + 100 = 101, 2 + 99 = 101, 3 + 8 =101,…, 50 + 51 = 101.

У початковій школі, де навчався математик Карл Фрідріх Гаусс (6 років) https://uk.wikipedia.org/wiki/Карл_Фрідріх_Гаусс, вчитель, щоб зайняти клас на тривалий час самостійною роботою, дав завдання учням – обчислити суму всіх натуральних чисел від 1 до 100. Маленький Гаус відповів на запитання майже миттєво, застосувавши вказаний спосіб підрахунку, чим неймовірно здивував усіх і, перш за все, вчителя.

Вирішення задач 14-24 з практичної роботи 4

Цикл while

Як уже було сказано попередньо, існують два основні різновиди циклу:

- цикли, що повторюються певну кількість разів (for, лічильні цикли, ***counting loops***);
- цикли, що повторюються до настання певної події (while, умовні цикли, ***conditional loops***)

Ми вже розглянули роботу циклу `for`, який є лічильним циклом. Цикл `for` чудово працює, якщо ми знаємо, скільки повторень (ітерацій) нам потрібно зробити. Але іноді потрібно, щоб цикл виконувався до настання деякої події, і кількість ітерацій у цьому випадку заздалегідь оцінити неможливо. І тут на допомогу приходить цикл `while.`

Структура циклу `while` у Python виглядає так:

```
while умова:
     блок коду
```

Двокрапка (:) в кінці рядка з інструкцією `while` повідомляє Python, що далі знаходиться ***блок команд***. До блоку входять усі рядки, розташовані з відступом від рядка з інструкцією `while`, аж до наступного рядка без відступу.

Блок команд, що виконується у циклі `while`, називається ***тілом циклу***.

Розглянемо код, який використовує цикл `while`, який роздрукує 10 разів слово `Привіт`:

```
i = 0
while i < 10:
     print('Привіт')
     i += 1
```

Такий код можна легко замінити циклом `for`, оскільки ми знаємо заздалегідь скільки разів потрібно виконати тіло циклу. Проте так не завжди.

Напишемо програму, яка зчитує числа та виводить їх квадрати, доки не буде введено -1. За такої постановки задачі ми можемо скористатися циклом `for`, оскільки не знаємо скільки чисел передуватиме числу -1.

```
num = int(input())
while num != -1:
```

```
    print('Квадрат вашого числа дорівнює:', num * num)
    num = int(input())
```

в якості початкового значення змінної num, ми використовуємо перше з чисел. Далі поки виконується умова циклу, саме, поки введене число не дорівнює -1, ми виконуємо тіло циклу. У тіло циклу входить дві команди:

1. надрукувати квадрат уведеного числа;
2. рахувати наступне число.

Важливим є два моменти:

1. правильна ініціалізація змінної num;
2. зміна змінної num всередині циклу while.

Важливо: якщо не змінювати змінну num всередині циклу, можна отримати так званий нескінченний цикл, який буде виконуватися нескінченно багато разів.

Цикл while дуже нагадує умовний оператор if. Різниця полягає в тому, що у випадку з умовним оператором відповідний блок коду буде виконуватися лише один раз, тоді як із циклом while блок коду буде виконано багаторазово.

Цикл for VS цикл while

Ми завжди можемо замінити цикл for за допомогою циклу while. Наступні дві програми виводять числа від 0 до 100:

```
# використовуємо for
for i in range(101):
    print(i)

# використовуємо while
i = 0
while i < 101:
    print(i)
    i += 1
```

У першому циклі змінна i послідовно набуває значень від 0 до 100. Для циклу while, нам довелося завести самостійно змінну i і надати їй початкового значення. Тіло циклу while містить аналогічну інструкцію виведення print(i), проте ми самостійно збільшуємо значення змінної i на 1, що робиться автоматично у випадку з циклом for.

Напишемо програму, що виводить усі числа, кратні 3, використовуючи цикл `for` і `while`:

```
# використовуємо for
for i in range(0, 100, 3):
     print(i)

# використовуємо while
i = 0
while i < 100:
     print(i)
     i += 3
```

Не завжди, проте вдається замінити цикл `while` за допомогою циклу `for`. Якщо заздалегідь невідомо кількість ітерацій, необхідно використовувати цикл `while` і тільки його.

Зчитування даних до стоп значення

Часто при вирішенні задач на цикл `while`, ми зчитуємо дані, доки користувач не введе деяке значення, яке називають ***стоп значенням***. Напишемо програму, яка зчитує числа і знаходить їх суму, доки користувач не введе слово `stop`:

```
text = input()
total = 0
while text != 'stop':
     num = int(text)
     total += num
     text = input()
print('Сума чисел дорівнює', total)
```

Такий код буде часто використовуватися під час вирішення задач.

Нескінечний цикл

Завжди, окрім поодиноких випадків, цикл `while` повинен містити можливість завершитися. Тобто в циклі щось має зробити умову, що перевіряється, хибною. Якщо цикл немає можливості завершитися, він називається ***нескінченним циклом***. Нескінченний цикл продовжує повторюватися доти, доки програма не буде перервана. Нескінченні цикли зазвичай з'являються, коли програміст забуває написати програмний код усередині циклу, який робить умову, що перевіряється, помилковою. Найчастіше слід уникати застосування нескінченних циклів.

Приклад нескінченного циклу:

```
i = 0
total = 0
while i < 10:
     total += i
```

Оскільки в тілі циклу немає зміни змінної `i`, то умова `i < 10` залишається істинною і цикл виконується нескінченно багато разів.

Нескінченні цикли можна використовувати у зв'язці з оператором переривання `break`.

Примітки

Примітка 1. Цикл `while` отримав свою назву через характер своєї роботи: він виконує якусь задачу доти, ***доки*** умова є істинною. Слово *while* англійською означає саме "доки".

Примітка 2. Цикл `while` називають циклом з ***передумовою***, оскільки виконанню тіла циклу передує перевірка умови (спочатку перевіряється умова, а потім виконується тіло циклу).

Примітка 3. Одноразове виконання тіла циклу називається ***ітерацією*** циклу.

Примітка 4. Цикл `while` може не виконатися жодного разу. Наприклад, наступний код:

```
i = -1
while i > 0:
     print('Hello world!')
```

не виведе текст, оскільки умова `i > 0` хибна від початку.

Примітка 5. Графічне уявлення циклу `while` має вигляд:

Примітка 6. Умова циклу `while`, як і умовному операторі `if`, може містити логічні операції `or, and, not`.

Обробка цифр числа

При вивченні цілих чисел (тип даних `int`), ми говорили про операцію цілісного поділу `//` та операцію знаходження залишку від поділу одного цілого числа на інше `%`. Використовуючи цикл `while` та дві даних операції, можна обробити цифри числа з довільною кількістю розрядів (цифр).

Нехай дано натуральне число `n`. Тоді:

- результатом операції `n % 10` є остання цифра числа;
- результатом операції `n // 10` є число з видаленою останньою цифрою.

Напишемо програму, яка зчитує натуральне число (ціле позитивне) та обробляє його цифри.

```
n = int(input())
while n != 0: # поки в числі є цифри
     last_digit = n % 10 # отримати останню цифру
     # код обробки останньої цифри
     n = n // 10 # видалити останню цифру з числа
```

Цикл `while` працює до тих пір, поки в числі є необроблені цифри. Тіло циклу містить:

1. процедуру отримання останньої цифри `last_digit = n%10`;
2. код обробки останньої цифри;
3. процедуру видалення останньої цифри з числа `n = n // 10`.

В якості процедури обробки може бути все, що завгодно: виведення цифр, знаходження суми, добутку цифр, знаходження найбільшої чи найменшої цифри, підрахунок цифр, що задовольняють певну умову і т.д.

Напишемо програму, яка визначає чи є в числі цифра 7.

```
num = int(input())
has_seven = False # сигнальна мітка

while num != 0:
     last_digit = num % 10
     if last_digit == 7:
         has_seven = True
     num = num // 10

if has_seven == True:
     print('YES')
else:
```

```
    print('NO')
```

Вирішення задач 25-38 з практичної роботи 4

Оператор переривання циклу break

Іноді буває потрібно ***перервати виконання циклу передчасно***. Оператор `break` перериває найближчий цикл `for` чи `while`.

Удосконалимо за допомогою оператора `break` програму, яка перевіряє число на простоту:

```
num = int(input())
flag = True

for i in range(2, num):
    if num % i == 0: # якщо вихідне число ділиться на якесь відмінне
від 1 і самого себе
        flag = False
        break # зупиняємо цикл якщо зустріли дільник числа

if flag: # еквівалентно if flag == True:
    print('Число просте')
else:
    print('Число складове')
```

Як тільки ми зустрічаємо дільник відмінний від 1 і `num`, ми змінюємо значення сигнальної мітки і переривається цикл, оскільки подальше його виконання не має сенсу: число гарантовано не є простим.

Оператор переривання циклу `break` дозволяє прискорювати програми, оскільки ми позбавляємось зайвих ітерацій.

Напишемо програму, що використовує цикл `for`, яка зчитує 10 чисел і підсумовує їх доти, доки не виявить негативне число. У цьому випадку виконання циклу переривається командою `break`:

```
result = 0
for i in range(10):
    num = int(input())
    if num < 0:
        break
    result += num
print(result)
```

Оператор переривання циклу `break` зручний у зв'язці з ***сигнальними мітками***: коли після перевірки деякої умови нам немає сенсу продовжувати виконання циклу.

Напишемо програму, яка визначає, чи містить введене користувачем число цифру 7.

```
num = int(input())
number = num
flag = False
while num != 0:
     last_digit = num % 10
     if last_digit == 7:
         flag = True
         break # перериваємо цикл, тому що число гарантовано містить
цифру 7
     num //= 10

if flag: # еквівалентно if flag == True:
     print('Число', number, 'містить цифру 7')
else:
     print('Число', number, 'не містить цифру 7')
```

Як тільки ми зустріли цифру 7, змінюємо значення сигнальної мітки і перериваємо цикл за допомогою оператора `break`. Ми можемо і не переривати цикл передчасно, а дочекатися його природного завершення (умова `num != 0`, тобто всі цифри числа оброблені), однак у такому випадку ми будемо виконувати зайву роботу, і якщо число дуже велике, програма буде працювати повільніше.

Нескінченні цикли

У попередніх темах ми говорили про цикл, який не може завершитися і назвали його ***нескінченним циклом***. Найпростіший спосіб створити нескінченний цикл у Python – записати наступний код:

```
while True:
     print('Python is awesome!')
```

Результатом виконання такого коду буде нескінченна кількість рядків тексту:

```
Python is awesome!
Python is awesome!
.
.
.
Python is awesome!
Python is awesome!
```

```
Python is awesome!
```

Нескінченний цикл продовжує повторюватися доти, доки програма не буде перервана. Вивчивши оператор `break`, ми отримали механізм переривання нескінченних циклів.

Можливо, вам може здатися, що нескінченні цикли не мають сенсу, проте це не зовсім так. Наприклад, ви можете написати програму, яка запускається та працює, постійно приймаючи запити на обслуговування. Програмний код такої програми може виглядати так:

```
while True:
     query = get_new_query() # отримуємо новий запит на обробку
     query.process() # обробляємо запит
```

Іноді за допомогою нескінченного циклу вдається зробити програмний код більш читабельним. Простішим може бути завершення циклу на основі умов усередині тіла циклу, а не на основі умов у його заголовку:

```
while True:
     if умова 1: # умова для зупинки циклу
         break
     ...
     if умова 2: # ще одна умова для зупинки циклу
         break
     ...
     if умова 3: # ще одна умова для зупинки циклу
         break
```

У таких випадках, коли існує безліч причин завершення циклу, часто їх простіше виділити з різних місць, ніж намагатися вказати всі умови завершення в заголовку циклу.

Важливо: нескінченні цикли можуть бути дуже корисними. Просто пам'ятайте, що ви повинні переконатися, що цикл у якийсь момент буде перерваний, щоб він справді не ставав нескінченним.

Оператор continue

Інша стандартна ідіома циклів – пропуск окремих елементів при переборі. Оператор `continue` дозволяє перейти до наступної ітерації циклу `for` чи `while` до завершення всіх команд у тілі циклу.

Напишемо програму, яка виводить усі числа від 1 до 100, крім чисел 7, 17, 29 та 78.

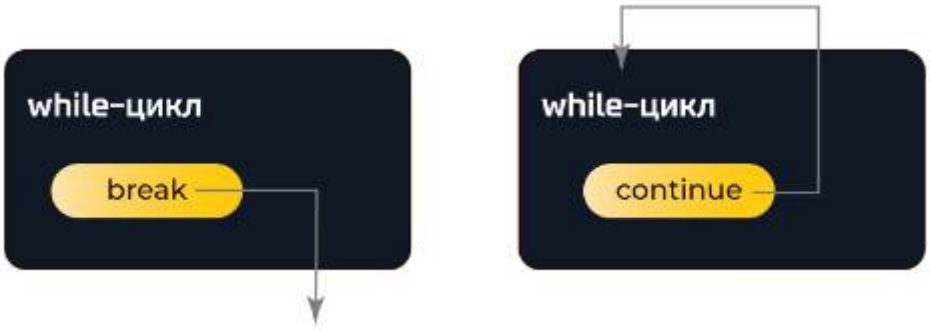
```
for i in range(1, 101):
```

```
    if i == 7 or i == 17 or i == 29 or i == 78:
        continue # переходимо на наступну ітерацію
    print(i)
```

Примітки

Примітка 1. Оператор `break` перериває виконання найближчого циклу, а не програми, тобто далі буде виконано команда, наступна відразу за циклом.

Примітка 2. Графічне подання операторів break і continue має вигляд:



Вирішення задач 39-40 з практичної роботи 4

Блок else у циклах

Python допускає необов'язковий блок `else` в кінці циклів `while` та `for`. Це унікальна особливість Python, яка не зустрічається в більшості інших мов програмування. Синтаксис такої конструкції наступний:

```
while умова:
    блок коду1
else:
    блок коду2

# або

for i in range(10):
    блок коду1
else:
    блок коду2
```

`Блок коду2`, вказаний в `else`, буде виконаний, коли ***штатним чином*** завершується цикл `while` або `for`.

Зараз ви можете подумати: "Як це може бути корисним?" Адже ми можемо зробити те саме, помістивши `блок коду2` відразу після циклу `while` або `for` без `else:`

```
while умова:
```

```
     блок коду1
блок коду2

# або

for i in range(10):
     блок коду1
блок коду2
```

В чому різниця?
Якщо слово `else` відсутнє в описі циклу, то `блок коду2` виконуватиметься після завершення циклу, незважаючи ні на що. Якщо ж слово `else` наявне, то `блок кода2` буде виконуватися тільки в тому випадку, якщо цикл завершується ***штатним чином***. Під ***штатним завершенням*** циклу мається на увазі його завершення без використання оператора переривання `break`.

Розглянемо наступний програмний код:

```
n = 5
while n > 0:
     n -= 1
     print(n)
else:
     print('Цикл завершений.')
```

Цей цикл повторюється до того часу, поки істинна умова `n > 0`. Оскільки цикл завершився ***штатним чином***, блок коду в інструкції `else` буде виконано. Таким чином, результатом виконання такої програми будуть рядки:

```
4
3
2
1
0
Цикл завершений.
```

Розглянемо наступний програмний код:

```
n = 5
while n > 0:
     n -= 1
     print(n)
     if n == 2:
         break
else:
     print('Цикл завершений.')
```

Цей цикл передчасно завершується за допомогою оператора переривання `break`, тому блок коду в інструкції `else` не буде виконано. Результатом виконання такої програми будуть рядки:

```
4
3
2
```

Вам може здатися, що інструкція в циклах `while` і `for` не зовсім відповідає тому, що реально відбувається. Гвідо ван Россум, творець Python, сказав, що якби він проектував мову Python заново, то позбавився б `else` в циклах.

Напишемо програму, яка визначає, чи містить введене користувачем число цифру 7. Замість програмного коду, написаного раніше:

```
num = int(input())
n = num
flag = False
while n != 0:
     last = n% 10
     if last == 7:
         flag = True
         break # перериваємо цикл, тому що число гарантовано містить
цифру 7
     n //= 10

if flag is True:
     print('Число', num, 'містить цифру 7')
else:
     print('Число', num, 'не містить цифру 7')
```

ми можемо використовувати:

```
num = int(input())
n = num
while n != 0:
     last = n% 10
     if last == 7:
         print('Число', num, 'містить цифру 7')
         break
     n //= 10
else:
     print('Число', num, 'не містить цифру 7')
```

Примітки

Примітка 1. Оператор `continue` не впливає на виконання блоку `else` у циклах.

Примітка 2. Блок `else` у циклах часто застосовується для обробки відсутності елементів.

Примітка 3. Блок коду `else` у циклах зустрічається не так часто на практиці. Однак якщо ви виявите ситуацію, в якій застосування `else` виправдане, то не соромтеся його використовувати. Це може додати ясності до вашого коду!

Рев'ю коду

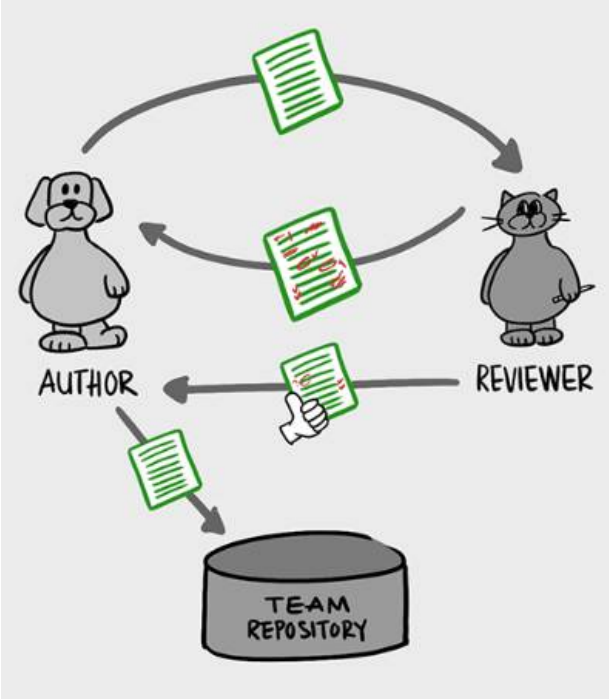
Рев'ю коду – перевірка вихідного коду програми з метою виявлення та виправлення помилок та неточностей, які залишилися непоміченими під час початкової розробки.

У процесі рев'ю коду можуть бути виправлені:

- фактичні помилки;
- продуктивність коду;
- читабельність коду та помилки форматування коду.

Метою рев'ю коду є ***покращення якості програмного коду*** та ***вдосконалення навичок програміста***.

Як правило, рев'ю коду виконує програміст з великим досвідом.



Фактичні помилки

До фактичних помилок у коді відносяться помилки, через які код може працювати неправильно. По суті, це помилки, що стосуються алгоритму, який використовується в програмі для вирішення задачі.

Серед частих фактичних помилок трапляються:

- відсутність початкової ініціалізації змінної;
- неправильна початкова ініціалізація змінної;
- відсутність відступу (у Python блоки коду виділяються відступами);
- неправильні числові граничні значення, наприклад, при використанні функції range();
- неправильні граничні порівняння (плутанина з >, >= або <, <=);
- плутанина логічних операцій or та and і т.д.

Продуктивність коду

Під продуктивністю коду у найпростішому випадку можна мати на увазі те, ***скільки часу програма витрачає на розв'язання задачі***. При написанні програми програміст повинен думати над тим, скільки часу ***в гіршому випадку*** буде потрібно його програмі для вирішення задачі.

Розглянемо задачу, яка перевіряє число на простоту.

1 версія програми: Перебираємо всі числа від `2` до `num - 1` і робимо перевірку ділимості числа `num` на `i`:

```
num = int(input())
flag = True

for i in range(2, num):
     if num % i == 0:
         flag = False
if num > 1 and flag == True:
     print('Число просте')
else:
     print('Число складове')
```

Якщо програмі на вхід подається просте число `num = 1934234249`, то вона працюватиме приблизно 270 секунд = 4.5 хвилини.

2 версія програми: Нескладно зрозуміти, що перебирати всі числа від `2` до `num - 1` не має сенсу. Достатньо перевірити числа від `2` до `num // 2 + 1`:

```
num = int(input())
flag = True

for i in range(2, num // 2 + 1):
     if num % i == 0:
```

```
        flag = False
if num > 1 and flag == True:
     print('Число просте')
else:
     print('Число складове')
```

Якщо програмі на вхід подається просте число `num = 1934234249`, вона працюватиме приблизно 130 секунд = 2.2 хвилини. ***По суті, ми покращили час роботи програми вдвічі!***

3 версія програми: Праву межу `num // 2 + 1` перевірки можна покращити, якщо помітити, що у будь-якого складового числа є дільник (відмінний від 1), що не перевищує ***квадратного кореня з числа***. Таким чином, має сенс перебирати дільники від 2 до $\sqrt{num}+1$.

```
num = int(input())
flag = True

for i in range(2, int(num ** 0.5) + 1):
     if num % i == 0:
         flag = False
if num > 1 and flag == True:
     print('Число просте')
else:
     print('Число складове')
```

Якщо програмі на вхід подається просте число `num = 1934234249`, вона працюватиме приблизно 0.066 секунд. ***По суті ми покращили час роботи програми у 4000 разів!***

4 версія програми: Попередні оптимізації стосувалися випадку, коли число, що перевіряється, є простим. Якщо число є складовим і ми знайшли його перший дільник (відмінний від 1), ми перериваємо цикл за допомогою оператора `break`:

```
num = int(input())
flag = True

for i in range(2, int(num ** 0.5) + 1):
     if num % i == 0:
         flag = False
         break
if num > 1 and flag == True:
     print('Число просте')
else:
     print('Число складове')
```

Читабельність коду

Слід пам'ятати, що код повинен легко читатися іншими програмістами. Щоб цього досягти, слід дотримуватися стандарту PEP 8.
https://peps.python.org/pep-0008/

Звертайте увагу на такі моменти:

1. ***відступи та пробіли***: використовуйте 4 пробіли на один рівень відступу та ніколи не змішуйте символи табуляції та пробіли;
2. ***назви змінних***: використовуйте назви, що говорять, для змінних (`total`, `counter`, `product`) і дотримуйтесь стилю ***lower_case_with_underscores*** (слова з маленьких букв з підкресленнями);
3. ***порожні рядки***: додаткові відступи порожніми рядками можуть бути зрідка корисні для виділення групи логічно пов'язаних частин програми: ініціалізація змінних, основний алгоритм, завершальна перевірка тощо;
4. ***коментарі***: коментарі мають бути закінченими реченнями. І пам'ятайте, коментарі, які суперечать коду, гірші, ніж відсутність коментарів. Завжди виправляйте коментарі, якщо ви змінюєте код!

Про стандарт PEP 8 українською мовою можна почитати тут:
https://devzone.org.ua/post/cistii-python-kod-osnovi

Примітки

Примітка 1. Програмні помилки часто називають ***багами***.
Причини виникнення дефектів і збоїв https://training.qatestlab.com/blog/technical-articles/causes-of-defects-and-failures/

Ціна помилки: найдорожчі комп’ютерні баги в історії IT
https://www.vectornews.net/news/society/66325-cna-pomilki-naydorozhch-kompyutern-bagi-v-storyi-t.html

Примітка 2. Всі програмісти створюють баги, особливо на початку свого кар'єрного шляху. Це норма.

Приклади

Приклад 1. Потрібно було написати програму, яка виводить символ A 10 разів. Ви рев'ювіте наступний код:

```
print('A')
print('A')
print('A')
print('A')
print('A')
print('A')
print('A')
print('A')
print('A')
print('A')
```

Що ви про нього скажете? Чи правильно він працює? Як його покращити?

Рев'ю. Наведений код виконує поставлену задачу, проте його можна покращити. Оскільки дії в рядках однакові, можна використовувати цикл. Оскільки ми знаємо кількість повторень (ітерацій), то підходить цикл `for`:

```
for _ in range(10):
     print('A')
```

Приклад 2. Потрібно написати програму, яка повинна вивести всі числа від 1 до 100 кратні 7. Ви рев'ювіте наступний код:

```
i = 1
while i < 101:
     if i % 7 == 0:
         print(i)
         i += 1
```

Що ви про нього скажете? Чи правильно він працює? Як його покращити?

Рев'ю. Наведений код містить досить поширену помилку: неправильно поставлений відступ. Оскільки управління циклом `while` відбувається за допомогою змінної `i`, її необхідно інкрементувати (збільшувати) кожну ітерацію. У наведеному коді вона інкрементується тільки якщо виконується умова `i % 7 == 0`, яка є хибною для початкового значення `i = 1`. Таким чином, отримуємо нескінченний цикл. Для виправлення помилки необхідно видалити відступ у рядка `i + = 1`:

```
i = 1
while i < 101:
     if i % 7 == 0:
         print(i)
     i += 1
```

В даному випадку краще використовувати цикл `for` з кроком, рівним 7. Це дозволить зробити код наочнішим і скоротить час виконання програми, оскільки немає необхідності переглядати і перевіряти всі числа на ділимість на 7:

```
for i in range(7, 101, 7):
     print(i)
```

Приклад 3. Потрібно було написати програму, яка виводить усі числа від 100 до 1 у порядку зменшення. Ви рев'ювіте наступний код:

```
for i in range(1, 100):
     print(101 - i)
```

Що ви про нього скажете? Чи правильно він працює? Як його покращити?

Рев'ю. Наведений код містить досить поширену помилку: неправильна права межа циклу `for`. Слід пам'ятати, що правий кордон у циклі `for` завжди не включний. Таким чином, для виправлення помилки необхідно замінити число 100 на 101:

```
for i in range(1, 101):
     print(101 - i)
```

Краще використовувати цикл `for` з кроком, рівним -1:

```
for i in range(100, 0, -1):
     print(i)
```

Приклад 4. Потрібно написати програму, яка знаходить суму всіх непарних чисел від 1 до 1000. Ви рев'ювіте наступний код:

```
a = 1
for i in range(1, 1000):
     if i % 2 == 1:
         a = a + 1
print(a)
```

Що ви про нього скажете? Чи правильно він працює? Як його покращити?

Рев'ю. Наведений код містить досить поширені помилки:

1. неправильна початкова ініціалізація змінної `a`;
2. неправильна права межа циклу `for;`
3. неправильно записана операція накопичування значення суми.

```
a = 0
for i in range(1, 1001):
     if i % 2 == 1:
         a = a + i
print(a)
```

Для покращення читабельності коду змінимо назву змінної `a` на `total` і використовуємо розширений оператор присвоєння:

```
total = 0
for i in range(1, 1001, 2):
     total += i
print(total)
```

Приклад 5. Потрібно було написати програму, яка обчислює факторіал числа. Ви рев'ювіте наступний код:

```
n = input()
a = 0
for i in range(n):
     a = a * i
print(a)
```

Що ви про нього скажете? Чи правильно він працює? Як його покращити?

Рев'ю. Наведений код містить досить поширені помилки:

1. відсутність перетворення рядка тексту на ціле число;
2. неправильна початкова ініціалізація змінної `a`;

3. неправильна робота з межами ітерування: змінна `i` набуває значень від 0 до n - 1.

```
n = int(input())
a = 1
for i in range(1, n + 1):
     a = a * i
print(a)
```

Для покращення читабельності коду, змінимо назву змінної `a` на `factorial` і використовуємо розширений оператор присвоєння:

```
n = int(input())
factorial = 1
for i in range(1, n + 1):
     factorial *= i
print(factorial)
```

У модулі `math` є функція `factorial()`, яка обчислює факторіал числа. Тому замість реалізації своєї версії, куди правильніше та зручніше скористатися вже готовою.

Вирішення задач 41-46 з практичної роботи 4

Вкладені цикли

Вкладений цикл розташований ще в одному циклі. Годинник є хорошим прикладом роботи вкладеного циклу. Секундна, хвилинна та годинна стрілки обертаються навколо циферблата. Годинна стрілка зміщується всього на 1 крок для кожних 60 кроків хвилинної стрілки. І секундна стрілка має зробити 60 кроків для 1 кроку хвилинної стрілки. Це означає, що для кожного повного оберту годинникової стрілки (12 кроків) хвилинна стрілка робить 720 кроків.

Розглянемо цикл, який частково моделює електронний годинник. Він показує секунди від 0 до 59:

```
for seconds in range(60):
     print(seconds)
```

Можна додати змінну `minutes` і вкласти цикл, написаний вище, в ще один цикл, який повторюється 60 разів:

```
for minutes in range(60):
     for seconds in range(60):
         print(minutes, ':', seconds)
```

Для того, щоб зробити модель закінченою, можна додати ще одну змінну для підрахунку годинника:

```
for hours in range(24):
     for minutes in range(60):
         for seconds in range(60):
             print(hours, ':', minutes, ':', seconds)
```

Результатом роботи такого коду буде:

```
0 : 0 : 0
0 : 0 : 1
0 : 0 : 2
...
23 : 59 : 58
23 : 59 : 59
```

Самий внутрішній цикл зробить 60 ітерацій для кожної ітерації середнього циклу. Середній цикл зробить 60 ітерацій для кожної ітерації самого зовнішнього циклу. Коли самий зовнішній цикл зробить 24 ітерації, середній зробить $24 \cdot 60 = 1440$ ітерацій, і внутрішній цикл зробить $24 \cdot 60 \cdot 60 = 86400$ ітерацій!

Приклад імітаційної моделі годинника підводить нас до кількох моментів, що стосуються вкладених циклів:

- вкладений цикл виконує всі свої ітерації для кожної окремої ітерації зовнішнього циклу;
- вкладені цикли завершують свої ітерації швидше, ніж зовнішні цикли;
- для того, щоб отримати загальну кількість ітерацій вкладеного циклу, треба ***перемножити*** кількість ітерацій всіх циклів.

Ми можемо вкладати один цикл в інший як `for`, так і `while`.

Оператори break і continue у вкладених циклах

Оператор `break` виконує переривання ***найближчого*** циклу, в якому він розташований. Аналогічно оператор `continue` здійснює перехід на наступну ітерацію ***найближчог***о циклу.

Розглянемо програмний код:

```
for i in range(3):
     for j in range(3):
         print(i, j)
```

Результатом його виконання будуть 9 рядків:

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

Змінимо код, додавши у вкладений цикл умовний оператор з оператором `break`:

```
for i in range(3):
     for j in range(3):
         if i == j:
             break
         print(i, j)
```

Результатом виконання нового коду будуть 3 рядки:

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

Змінимо оператор переривання `break` на оператор `continue`:

```
for i in range(3):
     for j in range(3):
         if i == j:
             continue
         print(i, j)
```

Результатом виконання нового коду будуть 6 рядків:

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

Якщо необхідно домогтися переривання зовнішнього циклу через виконання умови у внутрішньому, то варто зробити це через ***сигнальну мітку***.

Приклади розв'язання задач

Один цікавий спосіб дізнатися про роботу вкладених циклів полягає у їх використанні для виведення на екран комбінацій символів. Погляньмо на один простий приклад. Припустимо, що ми хочемо надрукувати на екрані зірочки у вигляді прямокутної таблиці:

```
******
******
******
******
******
******
******
******
```

Таблиця, що складається із зірочок, складається з 8 рядків та 6 стовпців. Наведений нижче фрагмент коду можна використовувати для виведення одного рядка зірочок:

```
for i in range(6):
     print ( '*', end='')
```

Для того, щоб завершити все виведення таблиці зірок, нам потрібно виконати цей цикл вісім разів. Ми можемо помістити цей цикл в ще один цикл, який робить вісім ітерацій, як показано нижче:

```
for i in range(8):
     for j in range(6):
         print('*', end='')
     print()
```

Зовнішній цикл робить вісім ітерацій. Під час кожної ітерації цього циклу внутрішній цикл робить 6 ітерацій. (Зверніть увагу, що у рядку 4 після того, як усі рядки були надруковані, ми викликаємо функцію `print()`. Ми повинні це зробити, щоб в кінці кожного рядка просунути екранний курсор на

наступний рядок. Без цієї інструкції всі зірочки будуть надруковані на екрані у вигляді одного довгого рядка.)

Давайте розглянемо ще один приклад. Припустимо, що ви хочете надрукувати зірочки в комбінації, яка схожа на наведений нижче зірковий трикутник:

```
*
**
***
****
*****
******
*******
********
```

Уявимо цю комбінацію зірочок, як поєднання рядків та стовпців. У цій комбінації всього вісім рядків. У першому рядку один стовпець. У другому рядку – два стовпці. У третьому рядку – три. І так триває до восьмого рядка, в якому вісім стовпців.

```
for i in range(8):
     for j in range(i + 1):
         print('*', end='')
     print()
```

Вирішення задач 47-51 з практичної роботи 4

Використання вкладених циклів при вирішенні рівнянь

Вкладені цикли можна використовувати для вирішення рівнянь з декількома змінними. Знаючи, що рішення (коріння) рівняння є натуральними (цілими) числами, нескладно написати програму, що містить вкладений цикл і перебирає всі можливі значення змінних.

Вирішення задач

Задача 1. Знайдіть усі пари натуральних чисел (і їх кількість), які є розв'язком рівняння $12x + 13y = 777$.

Рішення. Оскільки за умовою числа x та y є натуральними, то $x \le 64$, $y \le 59$ (при $x = 65$, $12 \cdot 65 = 780$, що більше, ніж 777, тому $x \le 64$, так як y не може бути негативним за умовою; аналогічно виходить, що $y \le 59$). Напишемо програму, яка перебирає всілякі пари чисел $(x; y)$ та перевіряє для них виконання умови $12x + 13y = 777$.

```
total = 0
for x in range(1, 65):
     for y in range(1, 60):
         if 12 * x + 13 * y == 777:
             total += 1
             print('x =', x, 'y =', y)
print('Загальна кількість натуральних рішень =', total)
```

В результаті виконання такого коду ми отримаємо:

```
x = 3 y = 57
x = 16 y = 45
x = 29 y = 33
x = 42 y = 21
x = 55 y = 9
Загальна кількість натуральних рішень = 5
```

Задача 2. Знайдіть усі пари натуральних чисел (і їх кількість), які є рішенням рівняння $x^2 + y^2 + z^2 = 2020$.

Рішення. Оскільки за умовою числа x, y та z є натуральними, то x, y, $z < \sqrt{2020} \approx 45$. Напишемо програму, яка перебирає всілякі трійки чисел (x; y; z) і перевіряє для них умову $x^2 + y^2 + z^2 = 2020$.

```
total = 0
for x in range(1, 45):
     for y in range(1, 45):
         for z in range(1, 45):
             if x ** 2 + y ** 2 + z ** 2 == 2020:
                 total += 1
                 print('x =', x, 'y =', y, 'z =', z)
print('Загальна кількість натуральних рішень =', total)
```

В результаті виконання такого коду ми отримаємо:

```
x = 18 y = 20 z = 36
x = 18 y = 36 z = 20
x = 20 y = 18 z = 36
x = 20 y = 36 z = 18
x = 36 y = 18 z = 20
x = 36 y = 20 z = 18
Загальна кількість натуральних рішень = 6
```

Вирішення задач 52-58 з практичної роботи 4

Лекція 5. Тип даних - рядки

Анотація. Рядковий тип даних. Основні операції над рядками, навчимося працювати з окремими символами, а також перебирати символи рядків. Вчимося робити рядкові зрізи, а також змінювати символи у рядку. Оновні методи конвертації регістру. Подання рядків у пам'яті комп'ютера, таблиці символів ASCII та Unicode.

План:

1. Індексація рядків
2. Ітерування рядків
3. Вирішення задач
4. Зрізи рядків
5. Зміна символів рядка
6. Вирішення задач
7. Методи конвертації регістру
8. Методи пошуку та заміни
9. Методи класифікації символів
10. Метод format
11. f-рядки
12. Подання рядків у пам'яті комп'ютера
13. Таблиця символів ASCII
14. Таблиця символів Unicode
15. Функція ord()
16. Функція chr()

Рядки в Python використовуються, коли треба працювати з текстовими даними.

Створення рядка. Для створення рядків ми використовуємо парні лапки ' або "":

```
s1 = 'Python'
s2 = "Pascal"
```

Зчитування рядка. Для зчитування текстових даних у рядкову змінну ми використовуємо функцію `input()`:

```
s = input() # вважали текст
num = int(input()) # вважали текст і перетворили його на ціле число
```

Порожній рядок. Для створення порожнього рядка ми пишемо `s = ''` або `s = ""`. Порожній рядок – це аналог числа 0.

Довжина рядка. Для визначення довжини рядка (кількості символів) ми використовуємо вбудовану функцію `len()`:

```
s = 'Hello'
n = len(s) # значення змінної дорівнює 5
print(n)
```

Конкатенація та множення на число. Оператори + та * можна використовувати для рядків. Оператор + зчіплює два і більше рядків. Це називається конкатенацією рядків. Оператор * повторює рядок вказану кількість разів.

Вираз	***Результат***
'AB' + 'cd'	'ABcd'
'A' + '7' + 'B'	'A7B'
'Hi'* 4	'HiHiHiHi'

Оператор приналежності in. За допомогою оператора in, ми можемо перевіряти, чи входить один рядок у склад інший. Тобто, чи є один рядок підрядком іншого:

```
s = 'All you need is love'
if 'love' in s:
     print('❤')
else:
     print('☻')
```

Так як рядок `s` містить підрядок `'love'`, то буде виведено смайлик ❤.

У Python можна використовувати смайли emoji https://pypi.org/project/emoji/

Індексація рядків

Дуже часто буває необхідно звернутись до конкретного символу у рядку. Для цього у Python використовуються квадратні дужки `[]`, у яких вказується індекс (номер) потрібного символу у рядку.

Нехай `s = 'Python'`. Таблиця нижче, показує, як працює індексація:

Вираз	***Результат***	***Пояснення***
s[0]	P	перший символ рядка
s[1]	y	другий символ рядка
s[2]	t	третій символ рядка
s[3]	h	четвертий символ рядка
s[4]	o	п'ятий символ рядка
s[5]	n	шостий символ рядка

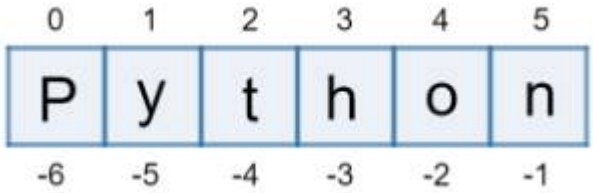
Зверніть увагу - перший символ рядка дорівнює s[0], а не s[1]. В Python індексація починається з 0, за аналогією з функцією `range(n)`, яка генерувала послідовність натуральних чисел від 0 до n - 1.

На відміну від багатьох мов програмування, у Python є можливість роботи з негативними індексами. Якщо перший символ рядка має індекс 0, то останньому елементу надається індекс -1.

Вираз	Результат	Пояснення
s[-6]	P	перший символ рядка
s[-5]	y	другий символ рядка
s[-4]	t	третій символ рядка
s[-3]	h	четвертий символ рядка
s[-2]	o	п'ятий символ рядка
s[-1]	n	шостий символ рядка

Таким чином, отримуємо

Позитивні індекси	0	1	2	3	4	5
Рядок	P	y	t	h	o	n
Негативні індекси	-6	-5	-4	-3	-2	-1



Часта помилка у програмістів-початківців — звернення за неіснуючим індексом у рядку.

Наприклад, якщо `s = 'Python'`, і ми спробуємо звернутися до `s[17]`, ми отримаємо помилку:

```
IndexError: string index out of range
```

Помилка виникає, оскільки рядок містить лише 6 символів.

Зверніть увагу: якщо довжина рядка `s` дорівнює `len(s)`, то при позитивній нумерації зліва направо, останній елемент має індекс рівний `len(s) - 1`, а при негативній індексації справа наліво, перший елемент має індекс рівний `-len(s).`

Ітерування рядків

Дуже часто потрібно просканувати весь рядок повністю, обробляючи кожен символ. Для цього зручно використати цикл `for`. Напишемо програму, яка виводить кожен символ рядка на окремому рядку:

```
s = 'abcdef'
for i in range(len(s)):
     print(s[i])
```

Результатом виконання такої програми будуть рядки:

```
a
b
c
d
e
f
```

Ми передаємо в функцію `range()` довжину рядка `len(s).` У нашому випадку довжина рядка `s` дорівнює 6. Таким чином, виклик функції `range(len(s))` має вигляд `range(6)` і змінна циклу `i` послідовно перебирає всі значення від 0 до 5. Це означає, що вираз `s[i]` послідовно поверне всі символи рядка `s`. Такий спосіб ітерації рядка зручний, коли нам потрібен не тільки сам елемент `s[i],` а й його індекс `i.`

Якщо нам не потрібен індекс самого символу, то ми можемо використовувати коротший спосіб ітерації:

```
s = 'abcdef'
for c in s:
     print(c)
```

Цей цикл пройде по рядку `s`, надаючи змінної циклу `c` значення кожного символу (!) на відміну від попереднього циклу, у якому змінна циклу «бігала» за індексами рядка.

Зверніть увагу на позначення змінних циклів. У першому циклі ми використовуємо ім'я `i`, що відповідає стандартній ідеології найменування змінних циклів. У другому циклі ми назвали змінну буквою `c` – перша буква слова char (символ).

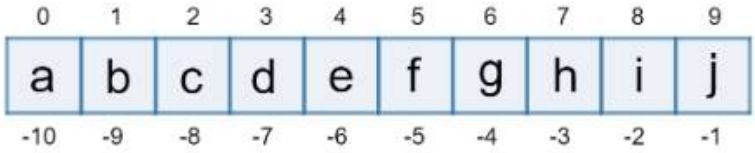
Вирішення задач 1-9 з практичної роботи 5

Зрізи рядків

У попередній темі ми навчилися працювати з конкретними символами рядка за допомогою індексів []. Іноді потрібно працювати з цілими частинами рядка, у такому випадку ми використовуємо ***зрізи (slices)***. Зрізи схожі на комбінацію індексації та функції `range()`.

Розглянемо рядок `s = 'abcdefghij'`.

Позитивні індекси	0	1	2	3	4	5	6	7	8	9
Рядок	a	b	c	d	e	f	g	h	i	j
Негативні індекси	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1



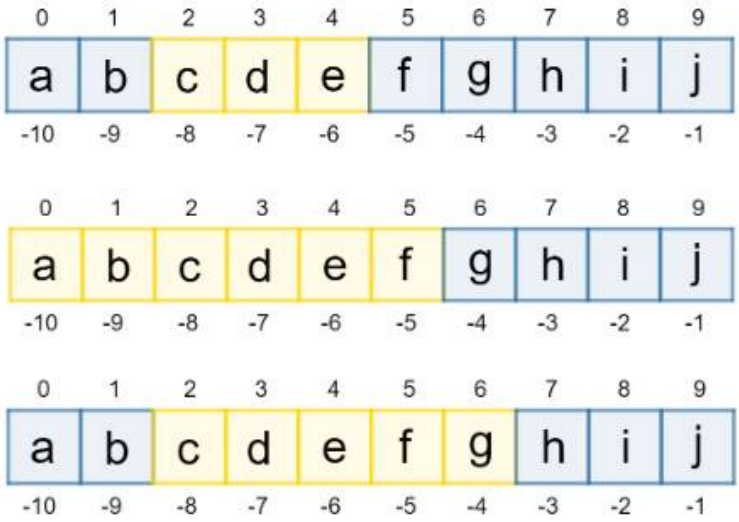
За допомогою зрізу ми можемо отримати кілька символів вихідного рядка, створивши діапазон індексів, розділених двокрапкою `s[x: y]`.

Наступний програмний код:

```
print(s[2:5])
print(s[0:6])
print(s[2:7])
```

виводить:

```
cde
abcdef
cdefg
```



При побудові зрізу `s[x:y]` перше число – це місце, де починається зріз (***включно***), а друге – це місце, де закінчується зріз (***невключно***). Розрізаючи рядки, ми створюємо підрядок, який по суті є рядком усередині іншого рядка.

Зріз до кінця, від початку

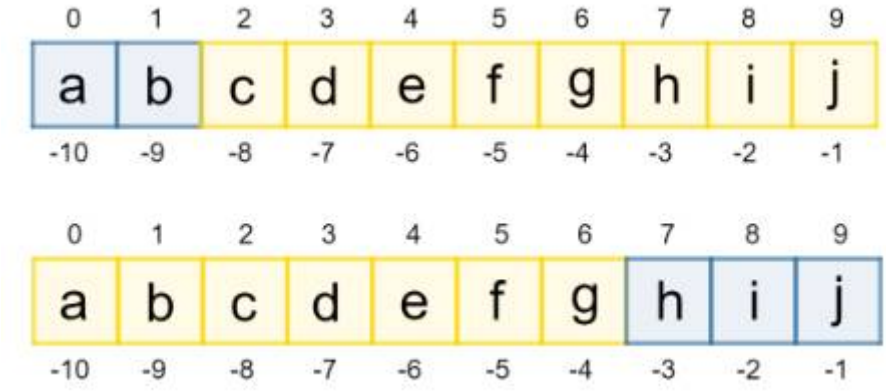
Якщо опустити другий параметр у зрізі `s[x:]` (але поставити двокрапку), то зріз береться до кінця рядка. Аналогічно, якщо опустити перший параметр `s[:y]`, то можна взяти зріз від початку рядка. Зріз `s[:]` збігається із самим рядком `s`.

Наступний програмний код:

```
print(s[2:])
print(s[:7])
```

виводить:

```
cdefghij
abcdefg
```



Зріз `s[:]` повертає вихідний рядок.

Негативні індекси у зрізі

Ми можемо також використовувати негативні індекси для створення зрізів. Як зазначалося раніше, негативні індекси рядка починаються з -1 і відраховуються до досягнення початку рядка. При використанні негативних індексів ***перший параметр зрізу повинен бути меншим за другий, або повинен бути пропущений***.

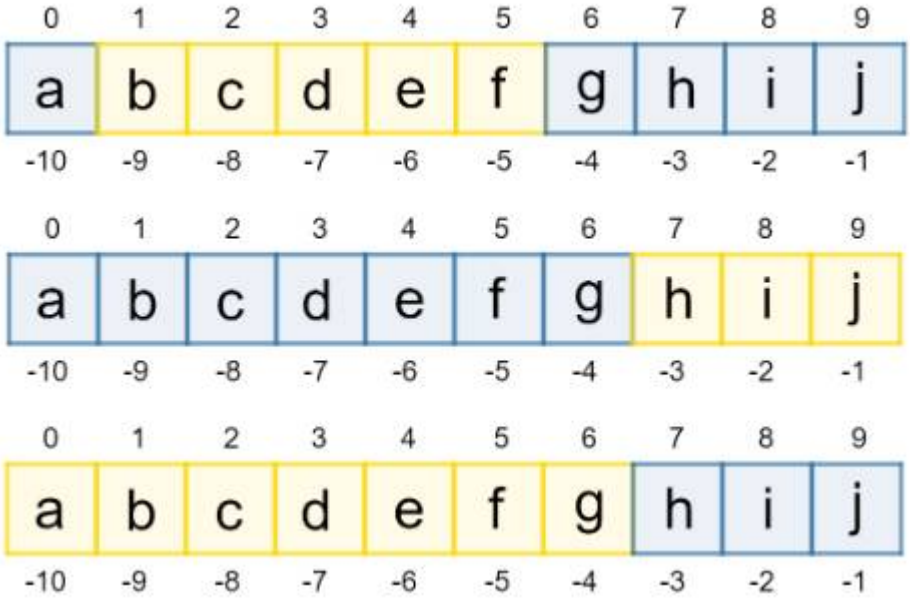
Наступний програмний код:

```
print(s[-9:-4])
print(s[-3:])
```

```
print(s[:-3])
```

виводить:

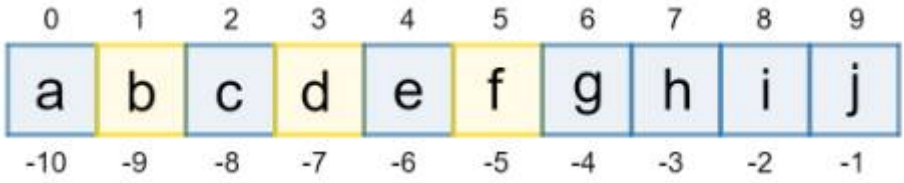
```
bcdef
hij
abcdefg
```



Видалити з рядка останній символ можна за допомогою зрізу `s[:-1]`.

Крок зрізу

Ми можемо передати до зрізу третій необов'язковий параметр, який відповідає за крок зрізу. Наприклад, зріз `s[1:7:2]` створить рядок `bdf`, що складається з кожного другого символу (індекси 1, 3, 5, правий кордон не включений у зріз).



Від’ємний крок зрізу

Якщо в якості кроку зрізу вказати ***від'ємне число***, то символи будуть йти у зворотному порядку.

Наступний програмний код:

```
print(s[::-1])
```

виводить:

```
jihgfedcba
```

Наступний програмний код:

```
print(s[1:7:2])
print(s[3::2])
print(s[:7:3])
print(s[::2])
print(s[::-1])
print(s[::-2])
```

виводить:

```
bdf
dfhj
adg
acegi
jihgfedcba
jhfdb
```

Підводячи підсумок

```
s = 'abcdefghij'
```

Програмний код	***Результат***	***Пояснення***
s[2:5]	cde	рядок складається з символів з індексами 2, 3, 4
s[:5]	abcde	перші п'ять символів рядка
s[5:]	fghij	рядок складається з символів з індексами від 5 до кінця
s[-2:]	ij	останні два символи рядка
s[:]	abcdefghij	весь рядок цілком
s[1:7:2]	bdf	рядок складається з кожного другого символу з індексами від 1 до 6
s[::-1]	jihgfedcba	рядок у зворотному порядку, тому що крок від'ємний

Зміна символу рядка за індексом

Припустимо, ми маємо рядок `s = 'abcdefghij'` і ми хочемо замінити символ з індексом 4 на 'X'. Можна спробувати написати код:

```
s[4] = 'X'
```

Однак такий код не працює. В Python рядки є ***незмінними***, тобто ми не можемо змінювати їх вміст за допомогою індексатора.

Якщо ми хочемо змінити символ рядка `s`, ми повинні створити новий рядок. Наступний код використовує зрізи та вирішує поставлене завдання:

```
s = s[:4] + 'X' + s[5:]
```

Ми створюємо два зрізи: від початку рядка до 5-го символу (не включно) та з 6-го символу (включно) до кінця рядка, а між ними вставляємо потрібний нам символ, який стане на 5-ту позицію (4 індекс).

Примітки

Примітка 1. Синтаксис рядків дуже схожий на синтаксис функції `range()`.
Примітка 2. Якщо перший параметр зрізу більший за другий, то зріз створює порожній рядок.

Вирішення задач 10-13 з практичної роботи 5

Методи та функції

Ми вже знайомі з деякими вбудованими функціями: `min(), max(), len(), int(), float()` тощо. Метод - спеціалізована функція, що тісно пов'язана з об'єктом. Як і функція, метод викликається для виконання окремої задачі, але він викликається для певного об'єкта і «знає» про свій цільовий об'єкт під час виконання.

Таким чином: метод – функція, що застосовується до об'єкта. Для нашого випадку - до рядка. Метод викликається у вигляді
`ім'я_об’єкта.ім’я_методу(параметры)`.

Наприклад, `s.find('e')` — це застосування до рядка `s` методу `find` з одним параметром `'e'`.

Методи рядкового типу даних можна поділити на три групи:

1. Конвертація регістру;
2. Пошук та заміна;
3. Класифікація символів.

Конвертація регістру

Методи цієї групи виконують перетворення регістру для рядків.

Метод capitalize()

Метод `capitalize()` повертає копію рядка `s`, у якому перший символ має верхній регістр, а решта символів мають нижній регістр.

Результатом виконання наступного коду:

```
s = 'foO BaR BAZ quX'
print(s.capitalize())
```

буде:

```
Foo bar baz qux
```

Символи, які не є літерами алфавіту, залишаються незмінними. Результатом виконання наступного коду:

```
s = 'foo123#BAR#.'
print(s.capitalize())
```

буде:

```
Foo123#bar#.
```

Метод swapcase()

Метод `swapcase()` повертає копію рядка `s`, де всі символи, що мають верхній регістр, перетворюються на символи нижнього регістру і навпаки.

Результатом виконання наступного коду:

```
s = 'FOO Bar 123 baz qUX'
print(s.swapcase())
```

буде:

```
foo bAR 123 BAZ Qux
```

Метод title()

Метод `title()` повертає копію рядка `s`, у якому перший символ кожного слова переводиться у верхній регістр.

Результатом виконання наступного коду:

```
s = 'the sun also rises'
print(s.title())
```

буде:

```
The Sun Also Rises
```

Цей метод використовує досить простий алгоритм: він намагається розрізнити важливі і погані слова і обробляє абревіатури і апострофи. Результатом виконання наступного коду:

```
s = "what's happened to ted's IBM stock?"
print(s.title())
```

буде:

```
What'S Happened To Ted'S Ibm Stock?
```

Метод lower()

Метод `lower()` повертає копію рядка `s`, де всі символи мають нижній регістр.

Результатом виконання наступного коду:

```
s = 'FOO Bar 123 baz qUX'
print(s.lower())
```

буде:

```
foo bar 123 baz qux
```

Метод upper()

Метод `upper()` повертає копію рядка `s`, у якому всі символи мають верхній регістр.

Результатом виконання наступного коду:

```
s = 'FOO Bar 123 baz qUX'
print(s.upper())
```

буде:

```
FOO BAR 123 BAZ QUX
```

Одне дуже важливе зауваження про методи цієї категорії полягає в тому, що вони не змінюють вихідний рядок. Якщо ви хочете змінити рядок `s`, необхідно написати код: `s = s.lower()`. Насправді тут ви створюєте зовсім інший об'єкт у пам'яті комп'ютера, просто він зі старою назвою `s`.

Вирішення задач 14-17 з практичної роботи 5

Пошук та заміна

Методи пошуку та заміни рядків усередині інших рядків.

Кожен метод у цій групі підтримує необов'язкові аргументи `<start>` та `<end>`. Як і в рядкових зрізах, дія методу обмежена частиною вихідного рядка, що починається з позиції символу `<start>` і триває аж до позиції символу `<end>`, але не включає її. Якщо параметр `<start>` вказано, а параметр `<end>` ні, то метод застосовується до частини вихідного рядка від `<start>` до кінця рядка. Якщо параметри не задані, мається на увазі, що `<start> = 0`, `<end> = len(s)`.

Метод count()

Метод `count(<sub>, <start>, <end>)` рахує кількість ***непересічних*** входів підрядка `<sub>` у вихідний рядок `s`.

Результатом виконання наступного коду:

```
s = 'foo goo moo'
print(s.count('oo'))
print(s.count('oo', 0, 8)) # підрахунок з 0 по 7 символ
```

буде:

```
3
2
```

Метод startswith()

Метод `startswith(<suffix>, <start>, <end>)` визначає чи ***починається*** вихідний рядок `s` підрядком `<suffix>`. Якщо вихідний рядок починається з підрядка `<suffix>`, метод повертає значення True, а якщо ні, то False.

Результатом виконання наступного коду:

```
s = 'foobar'
```

```
print(s.startswith('foo'))
print(s.startswith('baz'))
```

буде:

```
True
False
```

Метод endswith()

Метод `endswith(<suffix>, <start>, <end>)` визначає чи ***закінчується*** вихідний рядок `s` підрядком `<suffix>`. Метод повертає значення `True` якщо вихідний рядок закінчується на підрядок `<suffix>` і `False` інакше.

Результатом виконання наступного коду:

```
s = 'foobar'
print(s.endswith('bar'))
print(s.endswith('baz'))
```

буде:

```
True
False
```

Методи find(), rfind()

Метод `find(<sub>, <start>, <end>)` знаходить ***індекс першого входження*** підрядка `<sub>` у вихідному рядку `s`. Якщо рядок `s` не містить підрядка `<sub>`, метод повертає значення `-1`. Ми можемо використовувати цей метод нарівні з оператором `in` для перевірки: чи містить заданий рядок деякий підрядок чи ні.

Результатом виконання наступного коду:

```
s = 'foo bar foo baz foo qux'
print(s.find('foo'))
print(s.find('bar'))
print(s.find('qu'))
print(s.find('python'))
```

буде:

```
0
4
20
-1
```

Метод `rfind(<sub>, <start>, <end>)` ідентичний методу `find(<sub>, <start>, <end>)`, за тим винятком, що він шукає перше входження підрядка `<sub>` починаючи з кінця рядка `s`.

Методи index(), rindex()

Метод `index(<sub>, <start>, <end>)` ідентичний методу `find(<sub>, <start>, <end>)`, за винятком, що він ***викликає помилку*** `ValueError: substring not found` під час виконання програми, якщо підрядок `<sub>` не знайдено.

Метод `rindex(<sub>, <start>, <end>)` ідентичний методу `index(<sub>, <start>, <end>)`, за тим винятком, що він шукає перше входження підрядка `<sub>` починаючи з кінця рядка `s`.

Методи `find()` і `rfind()` є безпечнішими ніж `index()` і `rindex()`, оскільки не призводять до виникнення помилки під час виконання програми.

Метод strip()

Метод `strip()` повертає копію рядка `s`, у якого видалені всі пробіли, що стоять ***на початку і в кінці*** рядка.

Результатом виконання наступного коду:

```
s = '               foo bar foo baz foo qux                 '
print(s.strip())
```

буде:

```
foo bar foo baz foo qux
```

Метод lstrip()

Метод `lstrip()` повертає копію рядка `s`, у якого видалені всі пробіли, що стоять ***на початку*** рядка.

Результатом виконання наступного коду:

```
s = '     foo bar foo baz foo qux     '
print(s.lstrip())
```

буде:

```
foo bar foo baz foo qux␣ ␣ ␣ ␣ ␣
```

Метод rstrip()

Метод `rstrip()` повертає копію рядка `s`, у якого видалені всі пробіли, що стоять ***в кінці*** рядка.

Результатом виконання наступного коду:

```
s = '     foo bar foo baz foo qux     '
print(s.rstrip())
```

буде:

```
␣ ␣ ␣ ␣ ␣foo bar foo baz foo qux
```

Методи `strip()`, `lstrip()`, `rstrip()` можуть приймати на вхід опціональний аргумент `<chars>`. Необов'язковий аргумент `<chars>` – це рядок, який визначає набір символів для видалення.

Метод replace()

Метод `replace(<old>, <new>)` повертає копію `s` ***з усіма*** входженнями підрядка `<old>`, заміненими на `<new>`.

Результатом виконання наступного коду:

```
s = 'foo bar foo baz foo qux'
print(s.replace('foo', 'grault'))
```

буде:

```
grault bar grault baz grault qux
```

Метод `replace()` може приймати опціональний третій аргумент `<count>`, який визначає кількість замін.

Результатом виконання наступного коду:

```
s = 'foo bar foo baz foo qux'
print(s.replace('foo', 'grault', 2))
```

буде:

```
grault bar grault baz foo qux
```

Вирішення задач 18-25 з практичної роботи 5

Класифікація символів

Методи в цій групі класифікують рядок на основі символів, які він містить.

Метод isalnum().

Метод `isalnum()` визначає, чи складається рядок з літерно-цифрових символів. Метод повертає значення `True`, якщо вхідний рядок не порожній і складається ***лише*** з літерно-цифрових символів і `False` в іншому випадку.

Результат виконання наступного коду:

```
s1 = 'abc123'
s2 = 'abc$*123'
s3 = ''

print(s1.isalnum())
print(s2.isalnum())
print(s3.isalnum())
```

буде:

```
True
False
False
```

Метод isalpha()

Метод `isalpha()` визначає, чи складається рядок з алфавітних символів. Метод повертає `True`, якщо рядок не порожній і складається ***лише*** з алфавітних символів і `False` інакше.

Результат виконання наступного коду:

```
s1 = 'abcabc'
s2 = 'ABC123'
s3 = ''

print(s1.isalpha())
print(s2.isalpha())
print(S3.isalpha())
```

буде:

```
True
False
False
```

Метод isdigit()

Метод `isdigit()` визначає, чи складається рядок ***лише*** з цифрових символів. Метод повертає `True`, якщо рядок не порожній і складається ***лише*** з числових символів і `False` інакше.

Результат виконання наступного коду:

```
s1 = '1234567'
s2 = 'ABC123'
s3 = ''

print(s1.isdigit())
print(s2.isdigit())
print(s3.isdigit())
```

буде:

```
True
False
False
```

Метод islower ()

Метод `islower()` визначає, чи ***всі*** алфавітні символи в рядку є малими (мають нижній регістр). Метод повертає `True`, якщо всі символи алфавіту у вихідному рядку є малими, і `False` в іншому випадку. ***Всі неалфавитні символи ігноруються!***

Результат виконання наступного коду:

```
s1 = 'abc'
s2 = 'abc1$d'
s3 = 'abc1$d'

print(s1.islower())
print(s2.islower())
print(s3.islower())
```

буде:

```
True
True
False
```

Метод isupper()

Метод `isupper()` визначає, чи ***всі*** алфавітні символи в рядку є великими літерами (мають верхній регістр). Метод повертає `True`, якщо всі символи алфавіту у вихідному рядку є великими, і `False` в іншому випадку. ***Всі неалфавитні символи ігноруються!***

Результат виконання наступного коду:

```
s1 = 'ABC'
s2 = 'ABC1$D'
s3 = 'abc1$d'

print(s1.isupper())
print(s2.isupper())
print(s3.isupper())
```

буде:

```
True
True
False
```

Метод isspace()

Метод `isspace()` визначає, чи вихідний рядок складається ***лише*** з пробілів. Метод повертає `True`, якщо рядок складається лише з пробілів, і `False` в іншому випадку.

Результат виконання наступного коду:

```
s1 = ''
s2 = 'abc1$d'

print(s1.isspace())
print(s2.isspace())
```

буде:

```
True
False
```

Форматування рядків

Зберігати рядки в змінних зручно, але часто буває необхідно ***збирати рядки*** з інших об'єктів (рядків, чисел тощо) і виконувати з ними потрібні

маніпуляції. Для цього можна скористатися механізмом ***форматування рядків***.

Розглянемо наступний код:

```
age = 27
txt = 'My name is Irina, I am' + age
print(txt)
```

Такий код призводить до помилки під час виконання програми, оскільки ми намагаємось скласти число та рядок. Для вирішення такої проблеми ми можемо використовувати функцію `str`, яка перетворює числове значення в рядок:

```
age = 27
txt = 'My name is Irina, I am' + str(age)
print(txt)
```

Такий код працює, однак у Python кращим способом форматування вважається використання методу `format`. Попередню програму можна переписати у вигляді:

```
age = 27
txt = 'My name is Irina, I am {}'.format(age)
print(txt)
```

Ми передаємо необхідні параметри методу `format`, а Python форматує вказаний рядок та поміщає їх у рядок на місце заповнювачів `{}`. Ми можемо створювати скільки завгодно заповнювачів у рядку:

```
age = 27
name = 'Irina'
profession = 'math teacher'
txt = 'My name is {}, I am {}, I work as a {}'.format(name, age,
profession)
print(txt)
```

Для наочності та гнучкості форматування ми можемо використовувати порядковий номер у заповнювачі: `{0}, {1}, {2},...`. Такий номер визначає позицію параметра, переданого методу `format` (нумерація починається з нуля):

```
age = 27
name = 'Irina'
profession = 'math teacher'
txt = 'My name is {0}, I am {1}, I work as a {2}'.format(name, age,
profession)
```

```
print(txt)
```

Параметр `name` встає в {0} заповнювач, параметр `age` встає в {1} заповнювач і т.д. Ми можемо використовувати одне й те ж число у кількох заповнювачах

```
name = 'Irina'
txt = 'My name is {0}-{0}-{0}'.format(name)
print(txt)
```

Результатом виконання такого коду буде:

```
My name is Irina-Irina-Irina
```

f-рядки

Метод `format` добре справляється із задачею форматування рядків, проте якщо параметрів багато, то код може здатися трохи надмірним:

```
first_name = 'Irina'
last_name = 'Perova'
age = 27
profession = 'math teacher'
affiliation = 'ONMU'
print('Hello, {0} {1}. You are {2}. You are a {3}. You were a member
of {4}'
                .format(first_name,   last_name,   age,   profession,
affiliation))
```

У Python 3.6 з'явився новий різновид рядків - так звані f-рядки. Якщо поставити перед рядком префікс f, в заповнювачі можна буде включити код, наприклад ім'я змінної. Попередній код можна записати у вигляді:

```
first_name = 'Irina'
last_name = 'Perova'
age = 27
profession = 'math teacher'
affiliation = 'ONMU'
print(f'Hello, {first_name} {last_name}. You are {age}. You are a
{profession}. You were a member of {affiliation}')
```

На місце заповнювача {`first_name`} встає значення змінної `first_name`, місце заповнювача {`last_name`} встає значення змінної `last_name` тощо.

Примітки

Примітка 1. Детальна документація із форматуванням рядків.
https://docs.python.org/3/library/string.html#custom-string-formatting

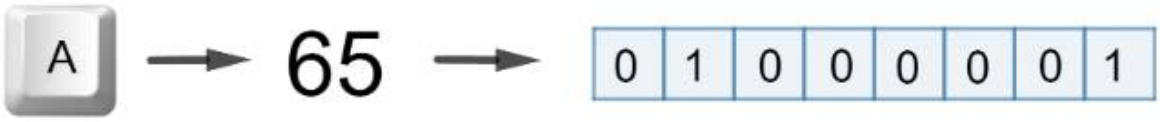
Подання рядків у пам'яті комп'ютера

Будь-який набір даних у оперативній пам'яті комп'ютера повинен зберігатися у вигляді двійкового числа. Це стосується і рядків, які складаються із символів (літери, розділові знаки тощо). Коли символ зберігається в пам'яті, він спочатку перетворюється на цифровий код. І потім цей цифровий код зберігається у пам'яті як двійкове число.

За минулі роки для представлення символів у пам'яті комп'ютера розробили різні схеми кодування. Історично найважливішою із цих схем кодування є схема кодування ASCII (American Standard Code for Information Interchange – американський стандартний код обміну інформацією).

Таблиця символів ASCII

ASCII являє собою набір з 128 цифрових кодів, які позначають англійські літери, різні розділові знаки та інші символи. Наприклад, код ASCII для великої англійської літери "A" (латинської) дорівнює 65. Коли на комп'ютерній клавіатурі ви набираєте літеру "A" у верхньому регістрі, в пам'яті зберігається число 65 (як двійкове число, зрозуміло).



Код ASCII для англійської "B" у верхньому регістрі дорівнює 66, для "C" у верхньому регістрі - 67 і т. д. ***На один символ в ASCII відводиться рівно 7 біт***.

Абревіатура ASCll вимовляється "аски".

ASCII table

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Набір символів ASCII був розроблений на початку 1960-х років і зрештою прийнятий майже всіма виробниками комп'ютерів. Однак схема кодування ASCII має обмеження, тому що визначає коди тільки для 128 символів. Для того, щоб це виправити, на початку 1990-х років було розроблено набір символів Юнікоду (Unicode). Це широка схема кодування, сумісна з ASCII, яка може представляти символи багатьох мов світу. Сьогодні Юнікод швидко стає стандартним набором символів, які використовуються в комп'ютерній індустрії.

Таблиця символів Unicode

Таблиця символів Юнікод являє собою набір цифрових символів, які включають знаки майже всіх письмових мов світу. Стандарт запропонований 1991 року некомерційною організацією «Консорціум Юнікоду». Застосування цього стандарту дозволяє закодувати дуже багато символів з різних систем писемності: у документах, закодованих за стандартом Юнікод, можуть сусідити китайські ієрогліфи, математичні символи, літери грецького алфавіту, латиниці та кирилиці, символи музичної нотації.

Стандарт складається з двох основних частин: універсального набору символів та сімейства кодувань (Unicode transformation format, UTF). Універсальний набір символів перераховує допустимі за стандартом Юнікод символи та надає кожному символу код у вигляді невід'ємного цілого числа. Сімейство кодувань визначає способи перетворення кодів символів для зберігання на комп'ютері та передачі.

У Юнікод весь час додаються нові символи, а сам розмір цієї таблиці не обмежений і тільки зростатиме, тому зараз при зберіганні в пам'яті одного юнікод-символу може знадобитися від 1 до 8 байт. Відсутність обмежень призвела до того, що почали з'являтися символи на всі випадки життя.

У Python рядки зберігаються у вигляді послідовності юнікод символів.

Примітки
Примітка 1. Офіційний веб-сайт таблиці символів Unicode.
https://home.unicode.org/

Примітка 2. Юнікод – це не кодування. Це таблиця символів. Те, як символи з відповідними кодами зберігатимуться в пам'яті комп'ютера, залежить від конкретного кодування, що базується на Юнікод, наприклад UTF-8.

Примітка 3. Перші 128 кодів таблиці символів Unicode збігаються з ASCII.

Функція ord

Функція `ord` дозволяє визначити код певного символу у таблиці символів Unicode. Аргументом цієї функції є одиночний символ.

Результатом виконання наступного коду:

```
num1 = ord('A')
num2 = ord('B')
num3 = ord('a')
print(num1, num2, num3)
```

буде:

```
65 66 97
```

Зауважте, що функція `ord` приймає саме ***одиночний символ***. Якщо спробувати передати рядок, який містить більше одного символу:

```
num = ord('Abc')
print(num)
```

ми отримаємо помилку часу виконання:

```
TypeError: ord() expected a character, але string of length 3 found
```

Назва функції `ord` походить від англійського слова order - порядок.

Функція chr

Функція chr дозволяє визначити за кодом символу сам символ. Аргументом цієї функції є чисельний код.

Результатом виконання наступного коду:

```
chr1 = chr(65)
chr2 = chr(75)
chr3 = chr(110)
print(chr1, chr2, chr3)
```

буде:

```
A K n
```

Назва функції chr походить від англійського слова char символ.

Функції ord і chr часто працюють у парі. Ми можемо використовувати наступний код для виведення всіх літер англійського алфавіту:

```
for i in range(26):
     print(chr(ord('A') + i))
```

Виклик функції ord('A') повертає код символу «А», який дорівнює 65. Далі на кожній ітерації циклу, до цього коду додається значення змінної i = 0, 1, 2, ..., 25, а потім отриманий код перетворюється в символ за допомогою виклику функції chr.

Примітки

Примітка. Функції ord та chr є ***взаємно зворотними***. Для них виконані рівності:

```
chr(ord('A')) = 'A', ord(chr(65)) = 65.
```

Вирішення задач 26-28 з практичної роботи 5

Лекція 6. Тип даних – списки. Основні алгоритми сортування

Анотація. Списки як збереження послідовностей та аналог масивів. Основи роботи зі списками. Додавання елементів до списку. Оператор del, що видаляє елементи за заданим індексом. Виведення елементів списку та рядка. Рядкові методи split() та join(). Інші методи списків. Спискові вирази. Створення списків без використання циклів і виклику спискового методу append().Задачі та способи (алгоритми) сортування списків.

План:

1. Створення списків
2. Порожні списки
3. Вбудована функція list()
4. Виведення списків
5. Вбудовані функції len(), sum(), min(), max()
6. Оператор приналежності in
7. Індексація та зрізи
8. Конкатенація та множення на число
9. Відмінність списків від рядків
10. Метод додавання елемента append()
11. Метод розширення списку extend()
12. Оператор del
13. Вирішення задач
14. Виведення списку за допомогою for
15. Виведення списку за допомогою розпакування
16. Виведення рядка за допомогою розпакування
17. Метод split()
18. Метод join()
19. Метод insert()
20. Метод index()
21. Метод remove()
22. Метод pop()
23. Метод reverse()
24. Метод count()
25. Метод clear()
26. Метод copy()
27. Метод sort()
28. Спискові вирази
29. Вирішення задач
30. Задача сортування
31. Сортування бульбашкою
32. Сортування вибором
33. Сортування простими вставками

Списки

У попередніх темах ми працювали з послідовностями чисел, символів, рядків, але не зберігали всю послідовність у пам'яті комп'ютера, а обробляли її поелементно, зчитуючи щоразу новий елемент. Однак у багатьох задачах потрібно ***зберігати всю послідовність***. Наприклад, класична задача сортування (упорядкування) деякої послідовності вимагає збереження всіх даних у пам'яті комп'ютера. На жаль, не зберігши їх неможливо відсортувати. І тут на допомогу приходить структура даних, яка в більшості мов програмування називається масивом. У Python вона називається ***списком***.

Структура даних (data structure) — програмна одиниця, що дозволяє ***зберігати та обробляти*** безліч однотипних та/або логічно пов'язаних даних.

Список є послідовністю елементів, пронумерованих від 0, як символи в рядку.

Створення списку

Щоб створити список, потрібно перерахувати його елементи через кому у квадратних дужках:

```
numbers = [2, 4, 6, 8, 10]
languages = ['Python', 'C#', 'C++', 'Java']
```

Список `numbers` складається з 5 елементів, і кожен із них — ***ціле число***.

```
numbers[0] == 2;
numbers[1] == 4;
numbers[2] == 6;
numbers[3] == 8;
numbers[4] == 10.
```

Список `languages` складається з 4 елементів, кожен з яких ***рядок.***

```
languages[0] == 'Python';
languages[1] == 'C#';
languages[2] == 'C++';
languages[3] == 'Java'.
```

Значення, укладені у квадратні дужки та відокремлені комами, називаються ***елементами списку***.

Список може містити значення ***різних типів даних***:

```
info = ['Marina', 1992, 61.5]
```

Список `info` містить рядкове значення, ціле число та число з плаваючою точкою.

```
info[0] == 'Marina';
info[1] == 1992;
info[2] == 61.5.
```

Порожній список

Створити порожній список можна двома способами:

1. Використовувати порожні квадратні дужки `[]`;
2. Використовувати вбудовану функцію, яка називається `list`.

Наступні два рядки коду створюють порожній список:

```
mylist = [] # порожній список
mylist = list() # теж порожній список
```

Виведення списку

Для виведення всього списку можна застосувати функцію `print()`:

```
numbers = [2, 4, 6, 8, 10]
languages = ['Python', 'C#', 'C++', 'Java']
print(numbers)
print(languages)
```

Функція `print()` виводить на екран елементи списку, у квадратних дужках, розділені комами:

```
[2, 4, 6, 8, 10]
['Python', 'C#', 'C++', 'Java']
```

Зверніть увагу, що виведення списку містить квадратні дужки. Пізніше ми навчимося виводити елементи списку в більш зручному вигляді за допомогою циклів або за допомогою розпакування.

Вбудована функція list

Python має вбудовану функцію `list()`, яка, крім створення порожнього списку, може перетворювати деякі типи об'єктів у списки.

Наприклад, ми знаємо, що функція `range()` створює послідовність цілих чисел у заданому діапазоні. Для перетворення цієї послідовності на список, ми пишемо наступний код:

```
numbers = list(range(5))
```

Під час виконання цього коду відбувається таке:

1. Викликається функція `range()`, в яку в якості аргументу передається число 5;
2. Ця функція повертає послідовність чисел `0, 1, 2, 3, 4`;
3. Послідовність чисел `0, 1, 2, 3, 4` передається як аргумент у функцію `list()`;
4. Функція `list()` повертає список `[0, 1, 2, 3, 4]`;
5. Список `[0, 1, 2, 3, 4]` присвоюється змінній `numbers`.

Ось ще один приклад:

```
even_numbers = list(range(0, 10, 2)) # список містить парні числа 0, 2, 4, 6, 8
odd_numbers = list(range(1, 10, 2)) # Список містить непарні числа 1, 3, 5, 7, 9
```

Так само за допомогою функції `list()` ми можемо створити список із символів рядка. Для перетворення рядка на список ми пишемо наступний код:

```
s = 'abcde'
chars = list(s) # список містить символи 'a', 'b', 'c', 'd', 'e'
```

Під час виконання цього коду відбувається таке:

1. Викликається функція `list()`, в яку в якості аргументу передається рядок `'abcde'`;
2. Функція `list()` повертає список `['a', 'b', 'c', 'd', 'e']`;
3. Список `['a', 'b', 'c', 'd', 'e']` присвоюється змінній `chars`.

Примітки

Примітка 1. Як уже було сказано, списки Python аналогічні масивам в інших мовах програмування. Однак різниця між списками і масивами все ж таки існує. Елементи масиву завжди мають однаковий тип даних і розташовуються в пам'яті комп'ютера безперервним блоком, а елементи списку можуть бути розкидані по пам'яті як завгодно і можуть мати різний тип даних.

Примітка 2. При виведенні вмісту списку за допомогою функції `print()` всі рядкові елементи списку обрамляються ***одинарними лапками***. Якщо потрібно здійснити виведення у подвійних лапках, потрібно самостійно писати код виведення.

Вирішення задач 1-2 з практичної роботи 6

Основи роботи зі списками

Робота зі списками дуже нагадує роботу з рядками, оскільки і списки, і рядки містять окремі елементи. Однак, елементи списку можуть мати довільний тип, а елементами рядків завжди є символи. Багато з того, що ми робили з рядками, є доступним і при роботі зі списками.

Функція len()

Довжиною списку називається кількість його елементів. Щоб порахувати довжину списку ми використовуємо вбудовану функцію `len()` (від слова length – довжина).

Наступний програмний код:

```
numbers = [2, 4, 6, 8, 10]
languages = ['Python', 'C#', 'C++', 'Java']

print(len(numbers)) # виводимо довжину списку numbers
print(len(languages)) # виводимо довжину списку languages

print(len(['apple', 'banana', 'cherry'])) # виводимо довжину списку, що складається з 3 елементів
```

виведе:

```
5
4
3
```

Оператор приналежності in

Оператор `in` дозволяє перевірити, чи містить список певний елемент.

Розглянемо наступний код:

```
numbers = [2, 4, 6, 8, 10]

if 2 in numbers:
     print('Список numbers містить число 2')
```

```
else:
     print('Список numbers не містить число 2')
```

Такий код перевіряє, чи містить список `numbers` число 2 і виводить відповідний текст:

```
Список numbers містить число 2
```

Ми можемо використовувати оператор `in` разом із логічним оператором `not`. Наприклад

```
numbers = [2, 4, 6, 8, 10]

if 0 not in numbers:
     print('Список numbers не містить нулів')
```

Індексація

Під час роботи з рядками ми використовували ***індексацію***, тобто звернення до конкретного символу рядка за його індексом. Аналогічно можна індексувати і списки.

Для індексації списків в Python використовуються квадратні дужки [], в яких вказується індекс (номер) потрібного елемента в списку:

Нехай `numbers = [2, 4, 6, 8, 10]`.

Таблиця нижче показує, як працює індексація:

Вираз	***Результат***	***Пояснення***
numbers[0]	2	перший елемент списку
numbers[1]	4	другий елемент списку
numbers[2]	6	третій елемент списку
numbers[3]	8	четвертий елемент списку
numbers[4]	10	п'ятий елемент списку

Зауважте: перший елемент списку `numbers[0]`, а не `numbers[1]`. Програмісти все рахують з нуля.

Так само, як і в рядках, для нумерації з кінця дозволено негативні індекси.

Вираз	***Результат***	***Пояснення***
numbers[-1]	10	п'ятий елемент списку
numbers[-2]	8	четвертий елемент списку
numbers[-3]	6	третій елемент списку

numbers[-4]	4	другий елемент списку
numbers[-5]	2	перший елемент списку

Як і в рядках, спроба звернутися до елемента списку за неіснуючим індексом:

```
print(numbers[17])
```

викличе помилку:

```
IndexError: index out of range
```

Зрізи

Розглянемо список `numbers = [2, 4, 6, 8, 10]`.

За допомогою зрізу ми можемо отримати кілька елементів списку, створивши діапазон індексів, розділених двокрапкою `numbers[x:y]`.

Наступний програмний код:

```
print(numbers[1:3])
print(numbers[2:5])
```

виводить:

```
[4, 6]
[6, 8, 10]
```

При побудові зрізу `numbers[x:y]` перше число – це місце, де починається зріз (***включно***), а друге – це місце, де закінчується зріз (***невключно***). Розрізаючи списки, ми створюємо нові списки, по суті підсписки вихідного.

При використанні зрізів зі списками ми можемо опускати другий параметр у зрізі `numbers[x:]` (але поставити двокрапку), тоді зріз береться до кінця списку. Аналогічно якщо опустити перший параметр `numbers[:y],` можна взяти зріз від початку списку.

Зріз `numbers[:]` повертає копію вихідного списку.

Як і в рядках, ми можемо використовувати негативні індекси у зрізах списків.

Використання зрізів для зміни елементів у заданому діапазоні

Для зміни цілого діапазону елементів списку можна використовувати зрізи. Наприклад, якщо ми хочемо перекласти українською мовою назви фруктів 'banana', 'cherry', 'kiwi', то це можна зробити за допомогою зрізу.

Наступний програмний код:

```
fruits = ['apple', 'apricot', 'banana', 'cherry', 'kiwi', 'lemon', 'mango']
fruits[2:5] = ['банан', 'вишня', 'ківі']

print(fruits)
```

виводить:

```
['apple', 'apricot', 'банан', 'вишня', 'ківі', 'lemon', 'mango']
```

Операція конкатенації + та множення на число *

Ми можемо використовувати оператори + і * для списків подібно до того, як ми це робили з рядками.

Наступний програмний код:

```
print([1, 2, 3, 4] + [5, 6, 7, 8])
print([7, 8] * 3)
print([0] * 10)
```

виводить:

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

Для генерації списків, що складаються строго з елементів, що повторюються, множення на число - найкоротший і правильний метод.

Ми також можемо використовувати розширені оператори += та *= під час роботи зі списками.

Наступний програмний код:

```
a = [1, 2, 3, 4]
b = [7, 8]
a += b # додаємо до списку a список b
```

```
b *= 5 # повторюємо список b 5 разів

print(a)
print(b)
```

виводить:

```
[1, 2, 3, 4, 7, 8]
[7, 8, 7, 8, 7, 8, 7, 8, 7, 8]
```

Вбудовані функції sum(), min(), max()

Вбудована функція `sum()` приймає в якості параметра список чисел і обчислює суму його елементів.

Наступний програмний код:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print('Сума всіх елементів списку =', sum(numbers))
```

виводить:

```
Сума всіх елементів списку = 55
```

Вбудовані функції `min()` і `max()` приймають в якості параметру список і знаходять мінімальний та максимальний елементи відповідно.

Наступний програмний код:

```
numbers = [3, 4, 10, 3333, 12, -7, -5, 4]
print('Мінімальний елемент =', min(numbers))
print('Максимальний елемент =', max(numbers))
```

виводить:

```
Мінімальний елемент = -7
Максимальний елемент = 3333
```

Відмінність списків від рядків

Незважаючи на всю схожість списків і рядків, є одна дуже важлива відмінність: рядки — ***незмінні*** об'єкти, а списки ***змінюються***.

Наступний програмний код:

```
s = 'abcdefg'
s[1] = 'x' # намагаємося змінити 2 символ (за індексом 1) рядка
```

призводить до помилки:

```
object does not support item assignment
```

Наступний програмний код:

```
numbers = [1, 2, 3, 4, 5, 6, 7]
numbers[1] = 101 # змінюємо 2 елемент (за індексом 1) списку
print(numbers)
```

виводить:

```
[1, 101, 3, 4, 5, 6, 7]
```

Запам'ятай: не можна змінювати окремі символи рядків, однак можна змінювати окремі елементи списків. Для цього використовуємо індексатор та оператор присвоювання.

Додавання елементів

Ми навчилися створювати статичні списки, тобто, списки, елементи яких відомі на етапі створення. Наступний крок – навчитися додавати елементи до вже існуючих списків.

Метод append()

Для додавання нового елемента ***до кінця списку*** використовується метод `append()`.

Наступний програмний код:

```
numbers = [1, 1, 2, 3, 5, 8, 13] # створюємо список

numbers.append(21) # додаємо число 21 до кінця списку
numbers.append(34) # додаємо число 34 до кінця списку

print(numbers)
```

виведе:

```
[1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Зверніть увагу, щоб використовувати метод `append()`, потрібно, щоб список був створений (при цьому він може бути порожнім).

Наступний програмний код:

```
numbers = [] # створюємо порожній список

numbers.append(1)
numbers.append(2)
numbers.append(3)

print(numbers)
```

виведе:

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

Важливо: ми не можемо використовувати індексатори для встановлення значень елементів списку, якщо список є порожнім. Наступний програмний код:

```
numbers = [] # створюємо порожній список

numbers[0] = 1
numbers[1] = 2
numbers[2] = 3

print(numbers)
```

призводить до помилки:

```
IndexError: list assignment index out of range
```

Метод extend()

Також можна розширити список іншим списком шляхом виклику методу `extend()`.

Наступний програмний код:

```
numbers = [0, 2, 4, 6, 8, 10]
odds = [1, 3, 5, 7]

numbers.extend(odds)
print(numbers)
```

виведе:

```
[0, 2, 4, 6, 8, 10, 1, 3, 5, 7]
```

Метод `extend()` розширює один список, додаючи щодо нього елементи іншого списку.

Відмінність між методами `append()` та `extend()` виявляється при додаванні рядка до списку.

Наступний програмний код:

```
words1 = ['iq option', 'stepik', 'beegeek']
words2 = ['iq option', 'stepik', 'beegeek']

words1.append('python')
words2.extend('python')

print(words1)
print(words2)
```

виведе:

```
['iq option', 'stepik', 'beegeek', 'python']
['iq option', 'stepik', 'beegeek', 'p', 'y', 't', 'h', 'o', 'n']
```

Метод `append()` додає рядок `'python'` повністю до списку, а метод `extend()` розбиває рядок `'python'` на символи `'p', 'y', 't', 'h', 'o', 'n'` та їх додає як елемент списку.

Видалення елементів

За допомогою оператора `del` можна видаляти елементи списку за певним індексом.

Наступний програмний код:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del numbers[5] # видаляємо елемент, що має індекс 5

print(numbers)
```

виведе:

```
[1, 2, 3, 4, 5, 7, 8, 9]
```

Елемент під вказаним індексом видаляється, а список перебудовується.

Зверніть увагу на синтаксис видалення, оскільки він відрізняється від звичайного виклику методу. При видаленні елементів не потрібно передавати аргумент усередині круглих дужок.

Оператор `del` працює і зі зрізами: ми можемо видалити цілий діапазон списків елементів.

Наступний програмний код:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del numbers[2:7] # видаляємо елементи з 2 по 6 включно

print(numbers)
```

виведе:

```
[1, 2, 8, 9]
```

Ми можемо видалити всі елементи на парних позиціях (0, 2, 4, ...) оригіналу.

Наступний програмний код:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]
del numbers[::2]

print(numbers)
```

виведе:

```
[2, 4, 6, 8]
```

Вирішення задач 3-11 з практичної роботи 6

Виведення елементів списку

При виведенні вмісту списку за допомогою функції `print()` виведення елементів здійснюється у квадратних дужках, причому всі елементи розділені комою. Таке виведення не завжди зручне, тому потрібно вміти виводити елементи списку у потрібний нам спосіб.

Виведення за допомогою циклу for

Для виведення елементів списку ***кожного на окремому рядку*** можна використовувати наступний код:

Варіант 1. Якщо потрібні індекси елементів:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
for i in range(len(numbers)):
     print(numbers[i])
```

Ми передаємо в функцію `range()` довжину списку `len(numbers)`. У нашому випадку довжина списку `numbers` дорівнює 11. Таким чином виклик функції `range(len(numbers))` має вигляд `range(11)` і змінна циклу `i` послідовно перебирає всі значення від 0 до 10. Це означає, що вираз `numbers[i]` послідовно поверне всі елементи списку `numbers`. Такий спосіб ітерації списку зручний, коли нам потрібний не тільки сам елемент `numbers[i]`, але його індекс `i`.

Варіант 2. Якщо індекси не потрібні:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for num in numbers:
     print(num)
```

Цей цикл пройде за списком `numbers`, надаючи змінній циклу `num` значення кожного елемента списку (!) на відміну попереднього циклу, у якому змінна циклу «бігала» за індексами списку.

Якщо потрібно виводити елементи списку на одному рядку, через пробіл, ми можемо використовувати необов'язковий параметр `end` функції `print()`:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

for num in numbers:
     print(num, end='')
```

Виведення за допомогою розпакування списку

Python має зручний спосіб виведення елементів списку без використання циклу `for`.

Варіант 1. Виведення елементів списку через один символ пробілу:

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(*numbers)
```

Такий код виведе:

```
0 1 2 3 4 5 6 7 8 9 10
```

Варіант 2. Виведення елементів списку кожного на окремому рядку

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(*numbers, sep='\n')
```

Такий код виведе:

```
0
1
2
3
4
5
6
7
8
9
10
```

Оскільки рядки містять символи, подібно до того, як списки містять елементи, то ми можемо використовувати розпакування рядків так само, як і розпакування списків.

Такий код:

```
s = 'Python'

print(*s)
print()
print(*s, sep='\n')
```

виведе:

```
P y t h o n

P
y
t
h
o
n
```

Вирішення задач 12-18 з практичної роботи 6

Рядкові методи

У попередньому модулі ми детально вивчили основні рядкові методи, проте обійшли стороною два важливі: `split()` і `join()`, які стосуються списків.

Вони протилежні за змістом: метод `split()` розбиває рядок по довільному роздільнику на список слів, а метод `join()` збирає рядок зі списку слів через заданий роздільник.

Метод split()

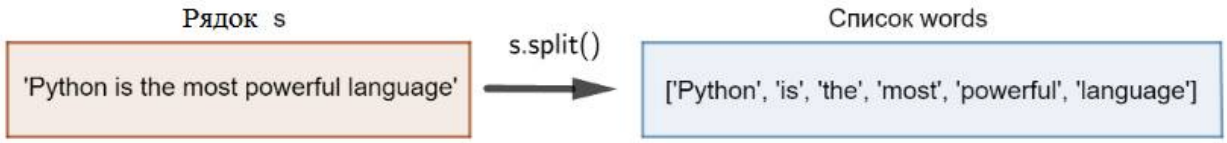
Метод `split()` розбиває рядок на слова, використовуючи в якості роздільника послідовність символів пробілу.

Наступний програмний код:

```
s = 'Python is the most powerful language'
words = s.split()
print(words)
```

виведе:

```
['Python', 'is', 'the', 'most', 'powerful', 'language']
```



Таким чином, виклик методу `split()` розбиває рядок на слова та повертає список, що містить усі слова.

Розглянемо наступний програмний код:

```
numbers = input().split()
```

Якщо під час запуску цієї програми ввести рядок `1 2 3 4 5`, то список `numbers` буде наступним `['1', '2', '3', '4', '5']`. Зверніть увагу, що список буде складатися з рядків, а не з чисел. Якщо потрібно отримати саме список чисел, то потім потрібно елементи списку по одному перетворити на числа:

```
numbers = input().split()
for i in range(len(numbers)):
     numbers[i] = int(numbers[i])
```

Необов'язковий параметр

Метод `split()` має необов'язковий параметр, який визначає, який набір символів буде використовуватися в якості роздільника між елементами

списку. Наприклад, виклик методу `split('.')` поверне список, отриманий поділом вихідного рядка за символом `'.'`.

Наступний програмний код:

```
ip = '192.168.1.24'
numbers = ip.split('.') # вказуємо явно роздільник
print(numbers)
```

виведе список:

```
['192', '168', '1', '24']
```

Метод join()

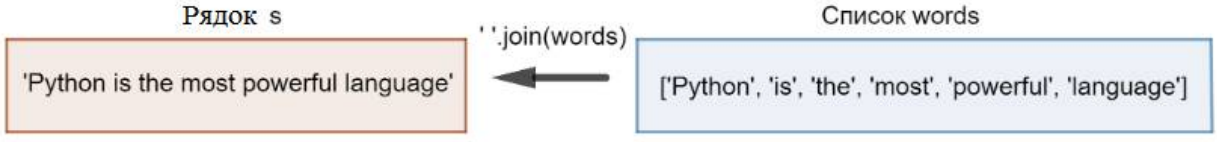
Метод `join()` збирає рядок з елементів списку, використовуючи в якості роздільника рядок, до якого застосовується метод.

Наступний програмний код:

```
words = ['Python', 'is', 'the', 'most', 'powerful', 'language']
s = ' '.join(words)
print(s)
```

виведе:

```
Python is the most powerful language
```



Зверніть увагу, всі слова розділені одним пробілом, оскільки метод `join()` викликався на рядку, що складається з одного символу пробілу `' '`.

Розглянемо ще кілька прикладів:

```
words = ['Ми', 'вчимо', 'мова', 'Python']
print('*'.join(words))
```

```
print('-'.join(words))
print('?'.join(words))
print('!'.join(words))
print('*****'.join(words))
print('abc'.join(words))
print('123'.join(words))
```

Результатом виконання такого коду буде:

```
Ми*вчимо*мову*Python
Ми-вчимо-мова-Python
Ми?вчимо?мова?Python
Ми! вчимо! мову! Python
Ми*****вчимо*****мову*****Python
МиabcучимаbcмоваabcPython
Ми123учимо123мова123Python
```

Запам'ятай: Строковий метод `split()` служить для перетворення рядка на список, а метод `join()` — на перетворення списку на рядок.

Примітки

Примітка 1. Існує велика різниця між результатами виклику методів `s.split()` та `s.split(' ')`. Різниця у поведінці проявляється, коли рядок містить кілька пробілів між словами.

Наступний програмний код:

```
s = 'Python is the most powerful language'
words1 = s.split()
words2 = s.split(' ')
print(words1)
print(words2)
```

виведе списки:

```
['Python', 'is', 'the', 'most', 'powerful', 'language']
['Python',  '',  '',  '',  'is',  '',  '',  'the',  '',  'most',  '',
'powerful', '', 'language']
```

Примітка 2. Методи `split()` та `join()` є рядковими методами. Наступний код призводить до помилки:

```
print([1, 2].split())
print([1, 2].join([3, 4, 5]))
```

Примітка 3. Строковий метод `join()` працює лише зі списком рядків. Наступний код призводить до помилки:

```
numbers = [1, 2, 3, 4] # Список чисел
s = '*'.join(numbers)
print(s)
```

Вирішення задач 19-25 з практичної роботи 6

Інші методи списків

1. Метод insert()
2. Метод index()
3. Метод remove()
4. Метод pop()
5. Метод reverse()
6. Метод count()
7. Метод clear()
8. Метод copy()
9. Метод sort()

Ми вже познайомилися з двома списковими методами `append()` та `extend()`. Перший додає до кінця списку один новий елемент, а другий розширює список іншим списком. До списків Python застосовні й інші зручні методи, з якими ми познайомимося в цій темі.

Метод insert()

Метод `insert()` дозволяє вставляти значення в список в заданій позиції. У нього передається два аргументи:

1. `index`: індекс, що визначає місце вставки значення;
2. `value`: значення, яке потрібно вставити.

Коли значення вставляється у список, список розширюється у розмірі, щоб розмістити нове значення. Значення, яке раніше знаходилося в заданій індексній позиції, і всі елементи після нього зсуваються на одну позицію до кінця списку.

Наступний програмний код:

```
names = ['Gvido', 'Roman', 'Timur']
print(names)
names.insert(0, 'Anders')
print(names)
names.insert(3, 'Josef')
print(names)
```

виведе:

```
['Gvido', 'Roman', 'Timur']
['Anders', 'Gvido', 'Roman', 'Timur']
['Anders', 'Gvido', 'Roman', 'Josef', 'Timur']
```

При вказівці неприпустимого індексу під час виконання програми помилки не відбувається. Якщо встановлений індекс за межами кінця списку, то значення буде додано до кінця списку. Якщо застосований негативний індекс, який вказує на неприпустиму позицію, значення буде вставлено на початок списку.

Метод index()

Метод `index()` повертає індекс першого елемента, значення якого дорівнює переданому в метод значенню. Таким чином, в метод передається один параметр:

1. `value`: значення, індекс якого потрібно знайти.

Якщо елемент у списку не знайдено, під час виконання відбувається помилка.

Наступний програмний код:

```
names = ['Gvido', 'Roman', 'Timur']
position = names.index('Timur')
print(position)
```

виведе:

```
2
```

Наступний програмний код:

```
names = ['Gvido', 'Roman', 'Timur']
position = names.index('Anders')
print(position)
```

призводить до помилки:

```
ValueError: 'Anders' is not in list
```

Щоб уникнути таких помилок, можна використовувати метод `index()` разом із оператором приналежності `in`:

```
names = ['Gvido', 'Roman', 'Timur']
if 'Anders' in names:
     position = names.index('Anders')
     print(position)
else:
     print('Такого значення немає у списку')
```

Метод remove()

Метод `remove()` видаляє перший елемент, значення якого дорівнює переданому в метод значенню. У метод передається один параметр:

1. `value`: значення, яке потрібно видалити.

Метод зменшує розмір списку на один елемент. Усі елементи після віддаленого елемента зміщуються на одну позицію до початку списку. Якщо елемент у списку не знайдено, під час виконання відбувається помилка.

Наступний програмний код:

```
food = ['Рис', 'Курица', 'Риба', 'Брокколі', 'Рис']
print(food)
food.remove('Рис')
print(food)
```

виведе:

```
['Рис', 'Курица', 'Риба', 'Брокколі', 'Рис']
['Курица', 'Риба', 'Брокколі', 'Рис']
```

Важливо: `remove()` видаляє лише перший елемент із зазначеним значенням. Усі наступні його входження залишаються у списку. Щоб видалити всі входження потрібно використовувати цикл `while` у зв'язці з оператором приналежності `in` та методом `remove`.

Метод pop()

Метод `pop()` видаляє елемент за вказаним індексом і повертає його. До методу `pop()` передається один ***необов'язковий*** аргумент:

1. `index`: індекс елемента, який потрібно видалити.

Якщо індекс не вказано, метод видаляє і повертає останній елемент списку. Якщо список порожній або вказаний індекс за межами діапазону, під час виконання відбувається помилка.

Наступний програмний код:

```
names = ['Gvido', 'Roman', 'Timur']
item = names.pop(1)
print(item)
print(names)
```

виведе:

```
Roman
['Gvido', 'Timur']
```

Метод count()

Метод `count()` повертає кількість елементів у списку, значення яких дорівнює переданому в метод параметру.

Таким чином, в метод передається один параметр:

1. `value`: значення, кількість входжень якого слід порахувати.

Якщо значення у списку не знайдено, метод повертає 0.

Наступний програмний код:

names = ['Timur', 'Gvido', 'Roman', 'Timur', 'Anders', 'Timur']

```
cnt1 = names.count('Timur')
cnt2 = names.count('Gvido')
cnt3 = names.count('Josef')

print(cnt1)
print(cnt2)
print(cnt3)
```

виведе:

```
3
1
0
```

Метод reverse()

Метод `reverse()` інвертує порядок проходження значень у списку, тобто змінює його на протилежний.

Наступний програмний код:

```
names = ['Gvido', 'Roman', 'Timur']
names.reverse()
print(names)
```

виведе:

```
['Timur', 'Roman', 'Gvido']
```

Існує велика різниця між викликом методу `names.reverse()` та використанням зрізу `names[::-1]`. Метод `reverse()` змінює порядок елементів на зворотний ***у поточному списку***, а зріз створює копію списку, у якому елементи йдуть у зворотному порядку.

Метод clear()

Метод `clear()` видаляє всі елементи зі списку.

Наступний програмний код:

```
names = ['Gvido', 'Roman', 'Timur']
names.clear()
print(names)
```

виведе:

```
[]
```

Метод copy()

Метод `copy()` створює поверхневу копію списку.

Наступний програмний код:

```
names = ['Gvido', 'Roman', 'Timur']
names_copy = names.copy() # створюємо поверхневу копію списку names

print(names)
print(names_copy)
```

виведе:

```
['Gvido', 'Roman', 'Timur']
['Gvido', 'Roman', 'Timur']
```

Аналогічного результату можна досягти за допомогою зрізів або функцій `list()`:

```
names = ['Gvido', 'Roman', 'Timur']
names_copy1 = list(names) # створюємо поверхневу копію за допомогою функції list()
names_copy2 = names[:] # створюємо поверхневу копію за допомогою зрізу від початку до кінця
```

Примітки

Примітка. Існує велика різниця в роботі рядкових та спискових методів. Рядкові методи не змінюють вмісту об'єкта, до якого вони застосовуються, а повертають нове значення. Спискові методи (крім методів `index()`, `count()`, `copy()`) навпаки змінюють вміст об'єкта, до якого застосовуються.

Наприклад, коли ми застосовуємо метод `lower()` до рядка `original_string`, то створюється новий рядок `lowercase_string`, який немає нічого спільного з рядком `original_string`. Точніше буде сказати, що рядковий метод повернув новий рядок, не змінюючи старий:

```
original_string = "HELLO WORLD"
lowercase_string = original_string.lower() # тут повертається новий, модифікований рядок

print(original_string) # HELLO WORLD
print(lowercase_string) # hello world
```

Як бачимо, сам рядок `original_string` не змінився. Ми використовували його для створення рядка `lowercase_string` через метод `lower()`. А `lowercase_string` справді створений на основі рядка `original_string`, де кожен символ у нижньому регістрі.

Зовсім інша ситуація виникає під час роботи зі списками. Наприклад, при додаванні елемента до списку за допомогою `append()` ми змінюємо наш список, не створюючи новий. Саме тому використання методу `append()` нічого не повертає.

```
original_list = [1, 2, 4, 7]
modified_list = original_list.append(666) # тут нічого не повертається

print(original_list) # [1, 2, 4, 7, 666]
print(modified_list) # None
```

Як бачимо, `original_list` змінився. Елемент 666 був доданий до кінця. А ось наша спроба присвоїти списку `modified_list` метод `append()` не увінчалася

успіхом. Повернено об'єкт `None`. Об'єкт `None` використовується для позначення відсутності значення або буквально " нічого " , тобто порожнечі.

Вирішення задач 26-29 з практичної роботи 6

Метод sort()

В Python списки мають вбудований метод `sort()`, який сортує елементи списку за зростанням або спаданням.

Наступний програмний код:

```
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99]
a.sort()
print('Відсортований список:', a)
```

виведе:

```
Відсортований список: [-67, -3, -2, 0, 1, 6, 7, 8, 9, 12, 34, 45, 99, 1000]
```

За замовчуванням метод `sort()` сортує список за зростанням. Якщо потрібно відсортувати список за спаданням, необхідно явно вказати параметр `reverse = True`.

Наступний програмний код:

```
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99]
a.sort(reverse=True) # сортуємо за спаданням
print('Відсортований список:', a)
```

виведе:

```
Відсортований список: [1000, 99, 45, 34, 12, 9, 8, 7, 6, 1, 0, -2, -3, -67]
```

Примітки

Примітка 1. За допомогою методу `sort()` можна сортувати списки, що містять не тільки числа, а й рядки. У такому разі елементи списку сортуються відповідно до лексикографічного порядку.
https://uk.wikipedia.org/wiki/Лексикографічний_порядок

Наступний програмний код:

```
a = ['бета', 'альфа', 'дельта', 'гамма']
```

```
a.sort()
print('Відсортований список:', a)
```

виведе:

```
'Відсортований список: ['альфа', 'бета', 'гамма', 'дельта']
```

Примітка 2. Метод `sort()` використовує алгоритм Timsort.
https://uk.wikipedia.org/wiki/Timsort

Вирішення задач 30 з практичної роботи 6

Створення списків

Для того, щоб створити список, що складається з 10 нулів ми можемо використовувати наступний код:

```
zeros = []
for i in range(10):
     zeros.append(0)
```

У Python, однак, є більш простий і компактний спосіб для створення такого списку. Ми можемо використовувати оператор множення списку на число:

```
zeros = [0] * 10
```

Для створення списків, заповнених за складнішими правилами, нам доводиться явно використовувати цикл `for`.

Наприклад, для створення списку цілих чисел від 0 до 9, ми змушені писати такий код:

```
numbers = []
for i in range(10):
     numbers.append(i)
```

Такий код хоч і не є складним, проте досить громіздкий.

Спискові вирази

Python має механізм для створення списків з неповторних елементів. Такий механізм називається ***списковим виразом*** (***list comprehension***).

Попередній код можна записати так:

```
numbers = [i for i in range(10)]
```

Загальний вигляд облікового виразу наступний:

```
[вираз for змінна in послідовність]
```

де `змінна` - ім'я деякої змінної, `послідовність` - послідовність значень, які вона набуває (список, рядок або об'єкт, отриманий за допомогою функції `range`), `вираз` - деякий вираз, як правило, залежить від використаної в списку змінної, яким будуть заповнені елементи списку .

Приклади використання спискових виразів

1. Створити список, заповнений 10 нулями можна і за допомогою спискового виразу:

```
zeros = [0 for i in range(10)]
```

2. Створити список, заповнений квадратами цілих чисел від 0 до 9, можна так:

```
squares = [i ** 2 for i in range(10)]
```

3. Створити список, заповнений кубами цілих чисел від 10 до 20, можна так:

```
cubes = [i ** 3 for i in range(10, 21)]
```

4. Створити список, заповнений символами рядка:

```
chars = [c for c in 'abcdefg']
print(chars)
```

Зчитування вхідних даних

При вирішенні багатьох задач із попередніх тем ми зчитували початкові дані (рядки, числа) та заповнювали ними список. За допомогою спискових виразів процес заповнення списку можна помітно скоротити.

Наприклад, якщо спочатку вводиться число *n* – кількість рядків, а потім самі рядки, то створити список можна так:

```
n = int(input())
lines = [input() for _ in range(n)]
```

Можна опустити опис змінної *n*:

```
lines = [input() for _ in range(int(input()))]
```

Якщо потрібно рахувати список чисел, необхідно додати перетворення типів:

```
numbers = [int(input()) for _ in range(int(input()))]]
```

Зверніть увагу, ми використовуємо символ _ як ім'я змінної циклу, оскільки вона не використовується.

Спискові вирази часто використовуються для ініціалізації списків. У Python не прийнято створювати порожні списки, а потім заповнювати їх значеннями, якщо цього можна уникнути.

Умови у списковому виразі

У списковий виразах можна використовувати умовний оператор. Наприклад, якщо потрібно створити список парних чисел від 0 до 20, ми можемо написати такий код:

```
evens = [i for i in range(21) if i % 2 == 0]
```

Важливо: щоб отримати список, що складається з парних чисел, краще використовувати функцію `range(0, 21, 2)`. Попередній приклад наведено для демонстрації можливості використання умов у спискових виразах.

Вкладені цикли

У списковому виразі можна використовувати вкладені цикли.

Наступний програмний код:

```
numbers = [i * j for i in range(1, 5) for j in range(2)]
print(numbers)
```

виведе список:

```
[0, 1, 0, 2, 0, 3, 0, 4]
```

Такий код рівнозначний наступному:

```
numbers = []

for i in range(1, 5):
     for j in range(2):
         numbers.append(i * j)
print(numbers)
```

Підводячи підсумок

Нехай `word = 'Hello', numbers = [1, 14, 5, 9, 12], words = ['one', 'two', 'three', 'four', 'five', 'six']`.

Список виразів	***Результуючий список***
`[0 for i in range(10)]`	`[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]`
`[i ** 2 for i in range(1, 8)]`	`[1, 4, 9, 16, 25, 36, 49]`
`[i * 10 for i in numbers]`	`[10, 140, 50, 90, 120]`
`[c * 2 for c in word]`	`['HH', 'ee', 'll', 'll', 'oo']`
`[m[0] for m in words]`	`['o', 't', 't', 'f', 'f', 's']`
`[i for i in numbers if i < 10]`	`[1, 5, 9]`
`[m[0] for m in words if len(m) == 3]`	`['o', 't', 's']`

Вирішення задач 31-39 з практичної роботи 6

Задача сортування

Задача сортування списку полягає у перестановці його елементів так, щоб вони були впорядковані за зростанням або зменшенням. Це одна з основних задач програмування. Ми стикаємося з нею дуже часто: під час запису прізвищ студентів у журналі, підбиття підсумків змагань тощо.

Алгоритми сортування

Алгоритм сортування – це алгоритм упорядкування елементів у списку. Алгоритми сортування оцінюються за швидкістю виконання та ефективності використання пам'яті:

- час - основний параметр, що характеризує швидкодію алгоритму;
- пам'ять — низка алгоритмів потребує виділення додаткової пам'яті під тимчасове зберігання даних.

Алгоритми сортування, які не споживають додаткової пам'яті, відносять до ***сортувань на місці***.

Основні алгоритми сортування

Повільні:

1. Пухирцеве сортування (Bubble sort);
2. Сортування вибором (Selection sort);
3. Сортування простими вставками (Insertion sort).

Швидкі:

1. Сортування Шелла (Shell sort);
2. Швидке сортування (Quick sort);
3. Сортування злиттям (Merge sort);
4. Пірамідальне сортування (Heap sort);
5. Сортування TimSort (використовується в Java та Python).

Більшість алгоритмів сортування, зокрема зазначені вище, засновані на порівнянні двох елементів списку. Існують однак алгоритми, не засновані на порівняннях. Такі алгоритми зазвичай використовують наперед задані умови щодо елементів списку. Наприклад, елементами списку є натуральні чи цілі числа у певному діапазоні, елементами є рядки тощо.

До алгоритмів не заснованих на порівняннях можна віднести такі:

1. Сортування підрахунком (Counting sort);
2. Блокове сортування (Bucket sort);
3. Порозрядне сортування (Radix sort).

В рамках курсу ми розглянемо нескладні алгоритми бульбашкового сортування, сортування вибором та сортування простими вставками.

Примітки

Примітка 1. Ми називаємо деякі алгоритми сортування ***повільними***, оскільки вони витрачають багато часу на сортування великих списків. Наприклад, якщо список містить близько мільйона елементів, то такі алгоритми витрачають години, а то й дні на виконання сортування, тоді як швидкі алгоритми справляються із задачею за секунди.

Примітка 2. Наочну роботу алгоритмів сортування різних вхідних даних можна побачити тут – File:Sorting quicksort anim.gif
https://commons.wikimedia.org/w/index.php?curid=1965827

Сортування бульбашкою

Алгоритм сортування бульбашкою складається з повторюваних проходів за списком, що сортується. За кожен прохід елементи послідовно порівнюються попарно і якщо порядок у парі неправильний, виконується обмін елементів. Проходи за списком повторюються $n-1$ раз, де n - довжина списку. При кожному проході алгоритму по внутрішньому циклу, черговий найбільший елемент списку ставиться на місце в кінці списку поряд з попереднім «найбільшим елементом».

Найбільший елемент щоразу «спливає» до потрібної позиції, як бульбашка у воді — звідси й назва алгоритму.

Алгоритм бульбашкового сортування вважається навчальним і практично не застосовується поза навчальної літератури, а на практиці застосовуються більш ефективні.

Розглянемо роботу алгоритму з прикладу сортування списку a = [5, 1, 4, 2, 8] за зростанням.

Перший прохід:

1. [5, 1, 4, 2, 8] → [1, 5, 4, 2, 8]: міняємо місцями перший і другий елементи, оскільки 5 > 1;
2. [1, 5, 4, 2, 8] → [1, 4, 5, 2, 8]: міняємо місцями другий і третій елементи, оскільки 5 > 4;
3. [1, 4, 5, 2, 8] → [1, 4, 2, 5, 8]: міняємо місцями третій та четвертий елементи, оскільки 5 > 2;
4. [1, 4, 2, 5, 8] → [1, 4, 2, 5, 8]: не міняємо четвертий і п'ятий елементи місцями, оскільки 5 < 8;
5. Найбільший елемент встав («сплив») на своє місце.

Другий прохід:

1. [1, 4, 2, 5, 8] → [1, 4, 2, 5, 8]: не міняємо перший і другий елементи місцями, оскільки 1 < 4;
2. [1, 4, 2, 5, 8] → [1, 2, 4, 5, 8]: міняємо місцями другий і третій елементи, оскільки 4 > 2;
3. [1, 2, 4, 5, 8] → [1, 2, 4, 5, 8]: не міняємо місцями третій та четвертий елементи, оскільки 4 < 5;
4. Другий за величиною елемент встав («сплив») на своє місце.

Тепер список повністю відсортований, але алгоритму це невідомо, і він працює далі.

Третій прохід:

1. [1, 2, 4, 5, 8] → [1, 2, 4, 5, 8]: не міняємо перший і другий елементи місцями, оскільки 1 < 2;
2. [1, 2, 4, 5, 8] → [1, 2, 4, 5, 8]: не міняємо другий і третій елементи місцями, оскільки 2 < 4;
3. Третій за величиною елемент встав («сплив») своє місце. (на якому і був)

Четвертий прохід:

1. [1, 2, 4, 5, 8] → [1, 2, 4, 5, 8]:
2. Четвертий за величиною елемент встав («сплив») своє місце.

Тепер список відсортований та алгоритм може бути завершений.

Реалізація алгоритму

Нехай потрібно відсортувати за зростанням список чисел: `a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99]`.

Наступний програмний код реалізує алгоритм бульбашкового сортування:

```
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99]
n = len(a)

for i in range(n - 1):
     for j in range(n - i - 1):
         if a[j] > a[j + 1]: # якщо порядок елементів пари
неправильний
             a[j], a[j + 1] = a[j + 1], a[j] # міняємо елементи пари
місцями

print('Відсортований список:', a)
```

Результатом виконання такого коду буде:

```
Відсортований список: [-67, -3, -2, 0, 1, 6, 7, 8, 9, 12, 34, 45, 99,
1000]
```

Оптимізація алгоритму

Алгоритм бульбашкового сортування можна трохи прискорити. Якщо на одному з чергових проходів виявиться, що обміни більше не потрібні, це означає, що всі елементи списку знаходяться на своїх місцях, тобто список відсортований. Для реалізації такого прискорення потрібно скористатися сигнальною міткою, тобто прапорцем та оператором переривання `break`.

Вирішення задачі 40 з практичної роботи 6

Сортування вибором

Сортування вибором покращує бульбашкове сортування, здійснюючи лише ***один обмін за кожен прохід за списком***. Для цього алгоритм шукає максимальний елемент та поміщає його на відповідну позицію. Як і для бульбашкового сортування, після першого проходу найбільший елемент знаходиться на правильному місці. Після другого проходу на місце стає

наступний максимальний елемент. Проходи за списком повторюються $n-1$ раз, де n – довжина списку, оскільки останній з них автоматично виявляється своєму місці.

Алгоритм сортування вибором також вважається навчальним та практично не застосовується поза навчальної літератури. На практиці використовують найефективніші алгоритми.

Розглянемо роботу алгоритму на прикладі сортування списку a = [5, 1, 8, 2, 4] за зростанням.

Перший прохід:

Знаходимо максимальний елемент 8 у невідсортованій частині списку та змінюємо його з останнім елементом списку:

[5, 1, 4, 2, 8].

Другий прохід:

Знаходимо максимальний елемент 5 у невідсортованій частині списку та змінюємо його з передостаннім елементом списку:

[2, 1, 4, 5, 8].

Третій прохід:

Знаходимо максимальний елемент 4 в невідсортованій частині списку і змінюємо його з передостаннім елементом списку:

[2, 1, 4, 5, 8].

Четвертий прохід:

Знаходимо максимальний елемент 2 у невідсортованій частині списку та змінюємо його з другим елементом списку:

[1, 2, 4, 5, 8].

Тепер список відсортований та алгоритм може бути завершений.

Замість максимального елемента можна шукати мінімальний.

Вирішення задачі 41 з практичної роботи 6

Сортування простими вставками

Алгоритм сортування простими вставками поділяє список на 2 частини - відсортовану та невідсортовану. З невідсортованої частини витягується черговий елемент і вставляється на потрібну позицію в відсортованій частині, в результаті чого відсортована частина списку збільшується, а невідсортована зменшується. Так відбувається, поки не вичерпано набір вхідних даних і не відсортовано всі елементи.

Сортування простими вставками є найбільш ефективним, коли список вже частково відсортований і елементів масиву небагато. Якщо елементів у списку менше 10, то цей алгоритм - один із найшвидших.

Розглянемо його роботу з прикладу сортування списку a = [5, 1, 8, 2, 4] за зростанням.

Перший прохід:

Ділимо список на дві частини: відсортовану [5] та невідсортовану [1, 8, 2, 4].

Виймаємо перший елемент 1 з невідсортованої частини списку і знаходимо йому місце у відсортованій частині:

[1, 5, 8, 2, 4].

Другий прохід:

Ділимо список на дві частини: відсортовану [1, 5] та невідсортовану [8, 2, 4].

Виймаємо перший елемент 8 з невідсортованої частини списку і знаходимо місце у відсортованій частині:

[1, 5, 8, 2, 4].

Третій прохід:

Ділимо список на дві частини: відсортовану [1, 5, 8] та невідсортовану [2, 4].

Виймаємо перший елемент 2 з невідсортованої частини списку і знаходимо місце у відсортованій частині:

[1, 2, 5, 8, 4].

Четвертий прохід:

Ділимо список на дві частини: відсортовану [1, 2, 5, 8] та невідсортовану [4].

Виймаємо перший елемент 4 з невідсортованої частини списку і знаходимо місце у відсортованій частині:

[1, 2, 4, 5, 8].

Тепер список відсортований і алгоритм може бути завершений.

Реалізація алгоритму

Нехай потрібно відсортувати за зростанням список чисел: `a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99]`.

Наступний програмний код реалізує алгоритм сортування простими вставками:

```
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6, 8, -2, 99]
n = len(a)

for i in range(1, n):
     elem = a[i] # беремо перший елемент із невідсортованої частини
списку
     j = i

     # поки елемент ліворуч існує і більше нашого поточного елемента
     while j >= 1 and a[j - 1] > elem:
         # зміщуємо j-й елемент відсортованої частини праворуч
         a[j] = a[j - 1]
         # самі йдемо вліво, далі шукаємо місце для нашого поточного
елемента
         j -= 1

     # знайшли місце для поточного елемента з невідсортованої частини
     # і вставляємо його на індекс j у відсортованій частині
     a[j] = elem


print('Відсортований список:', a)
```

Результатом виконання такого коду буде:

```
Відсортований список: [-67, -3, -2, 0, 1, 6, 7, 8, 9, 12, 34, 45, 99,
1000]
```

Оптимізація алгоритму

Алгоритм сортування простими вставками можна значно прискорити, якщо шукати потрібну позицію для вставки чергового елемента з невідсортованої частини списку за допомогою бінарного пошуку.
https://uk.wikipedia.org/wiki/Двійковий_пошук

Лекція 7. Функції користувача

Анотація. Створення функцій користувача без параметрів. Створення функцій користувача з параметрами. Локальні та глобальні змінні. Функції можуть приймати і не приймати параметри, але й повертати значення. Розглянемо цей особливий тип функцій – із поверненням значення. Крім функцій, що повертають числові значення, функції можуть повертати логічні, рядкові та інші значення. Функції з поверненням кількох значень. Переваги використання функцій.

План:

1. Функції без параметрів
2. Переваги використання функцій
3. Ключове слово pass
4. Вирішення задач
5. Функції з параметрами
6. Область видимості параметричної змінної
7. Вирішення задач
8. Локальні змінні
9. Область дії локальної змінної
10. Глобальні змінні
11. Глобальні константи
12. Вирішення задач
13. Функції із поверненням значення
14. Вирішення задач
15. Функції із поверненням значення
16. Вирішення задач
17. Функції із поверненням кількох значень
18. Вирішення задач
19. Переваги використання функцій

Функції

У попередніх темах ми використовували вбудовані в Python функції `print()`, `input()`, `int()`, `str()`, `len()` та багато інших. Настав час почати писати свої власні функції.

На самому початку курсу вам було запропоновано вирішити задачу, в якому потрібно було зобразити зірковий прямокутник розмірами 5×7 (5 рядків та 7 стовпців).

Наш перший варіант коду виглядав приблизно так:

```
print('*******')
print('*******')
print('*******')
```

```
print('*******')
print('*******')
```

Далі ми вивчили оператор множення рядка на число (оператор повторення) та написали б код:

```
print('*' * 7)
print('*' * 7)
print('*' * 7)
print('*' * 7)
print('*' * 7)
```

Ну і нарешті, ми вивчили цикли, після чого код набув би вигляду:

```
for _ in range(5):
     print('*' * 7)
```

А тепер уявімо, що таких прямокутників потрібно зобразити не один, а кілька, скажімо, 3 штуки.

Тоді код програми матиме вигляд:

```
for _ in range(5):
     print('*' * 7)

print()

for _ in range(5):
     print('*' * 7)

print()

for _ in range(5):
     print('*' * 7)
```

Результатом виконання такого коду буде:

```
*******
*******
*******
*******
*******

*******
*******
*******
*******
*******
```

```
*******
*******
*******
*******
*******
```

І хоча попередній код повністю вирішує поставлену задачу, він не позбавлений недоліків. По-перше, він доволі громіздкий через ***повторення частини коду***, що відповідає за виведення прямокутника. По-друге, якщо потрібно змінити розміри прямокутника, доведеться змінювати їх тричі, у кожній частині коду, що виводить прямокутник.

Замість повторення коду для виведення прямокутника можна перенести його в окрему ***функцію*** і викликати її 3 рази.

Для створення функції пишемо такий код:

```
def draw_box():
     for _ in range(5):
         print('*' * 7)
```

Коли функцію створено, щоб побачити результат її роботи, треба викликати її на ім'я:

```
draw_box()
```

Тепер, щоб зобразити 3 прямокутники, можна написати код:

```
draw_box()
print()
draw_box()
print()
draw_box()
```

Код став коротшим, читабельнішим (за рахунок вдалої назви функції), а головне, якщо будуть потрібні інші розміри прямокутника, достатньо буде змінити тільки саму функцію `draw_box()`.

Найменування функцій

Імена функцій призначаються так само, як змінним. Ім'я функції має бути досить описовим, щоб будь-який розробник, що читає код, міг здогадатися, що саме робить функція.

Python і тут вимагає дотримання тих самих правил, що при іменуванні змінних:

1. у імені функції використовуються лише латинські літери a-z, A-Z, цифри та символ нижнього підкреслення (_);

2. ім'я функції не може починатися з цифри;

3. ім'я функції по можливості має відображати її призначення;

4. символи верхнього та нижнього регістру різняться.

Пам'ятай: Python - регістрочутлива мова. Для назви змінних і функцій прийнято використовувати стиль ***lower_case_with_underscores*** (слова з маленьких літер з підкресленнями).

Оскільки функції ***виконують дії,*** більшість програмістів використовує в іменах функцій ***дієслова***. Наприклад:

- функцію, яка ***малює прямокутник***, можна назвати `draw_box();`
- функцію, яка ***друкує чек*** можна назвати `print_check();`
- функцію, яка обчислює заробітну плату до утримань, можна назвати `calculate_gross_pay().`

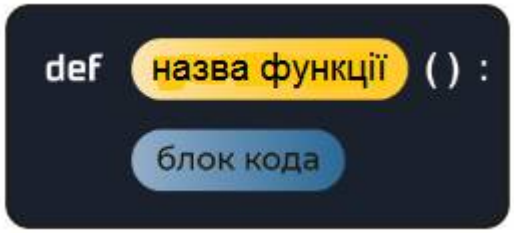
Кожне з цих імен дає опис того, що функція робить.

Оголошення функції

Отже, ***функція – окрема, функціонально незалежна частина програми, яка виконує певну задачу***.

Функції оголошуються з допомогою ключового слова `def` (від англ. define – визначати). За ключовим словом `def` слідують назва функції, круглі дужки `()` та двокрапка `:.`

```
def назва_функції():
    блок коду
```



Перший рядок оголошення функції називається ***заголовком функції***.

З наступного рядка йде блок коду – ***тіло функції***. Це набір інструкцій, що становлять одне ціле та виконуються щоразу, коли викликається функція.

Зверніть увагу, що кожен рядок у тілі функції виділено відступом.

Для виділення рядків блоку коду відступом програмісти Python зазвичай використовують ***чотири пробіли*** відповідно до стандарту PEP 8.

Розглянемо оголошення функції:

```
def print_message():
     print('Я - Артур,')
     print('король британців.')
```

Цей фрагмент коду визначає функцію під назвою `print_message().` Тіло її складається з двох інструкцій, і виклик призведе до їхнього виконання.

Виклик функції

Для виклику функції пишуть її назву та круглі дужки.

Важливо: дуже часто початківці програмісти забувають викликати функцію. Пам'ятайте, що оголошення функції не викликає її.

```
# оголошення функції
def print_message():
     print('Я - Артур,')
     print('король британців.')


# виклик функції
print_message()
```

Примітки

Примітка 1. Зверніть увагу, що оголошення функції користувача трохи нагадує використання умовного оператора `if` і циклів `for, while`.

Примітка 2. Оголошення функції має ***передувати*** її виклику. Наступний програмний код:

```
# виклик функції
print_message()

# оголошення функції
```

```
def print_message():
     print('Я - Артур,')
     print('король британців.')
```

призведе до помилки:

```
NameError: name 'print_message' is not defined
```

Примітка 3. При оголошенні функції слід переконатися, що кожен рядок функції починається з однакової кількості пробілів, інакше відбудеться помилка. Наприклад, наведене нижче визначення функції призведе до помилки:

```
def print_greeting():
     print('Доброго ранку!')


print('Сьогодні ми вивчатимемо функції.')
     print('Це дуже важлива тема!')
```

Примітка 4. Іноді, при оголошенні функції, потрібно зробити свого роду заглушку, щоб функція нічого не виконувала. Тоді ми використовуємо ключове слово `pass`:

```
def do_nothing():
     pass
```

Ми оголосили функцію під назвою `do_nothing()`. Тіло такої функції містить єдиний рядок коду, який нічого не робить.

Примітка 5. Функції часто називають ***підпрограмами***.

Вирішення задач 1-2 з практичної роботи 7

Функції з параметрами

У попередній темв ми визначили функцію `draw_box()`, яка виводить зірковий прямокутник із розмірами 5×7:

```
def draw_box():
     for i in range(5):
         print('*' * 7)
```

Було б набагато зручніше, якби функція `draw_box()` виводила прямокутник із довільними розмірами. Функції можуть приймати вхідні параметри, що робить їх більш гнучкими.

Функції з параметрами оголошуються наступним чином:

```
def назва_функції(параметри):
     блок коду
```

Давайте перепишемо попередню версію функції `draw_box()` так, щоб вона приймала параметри, що задають висоту та ширину прямокутника:

```
def draw_box(height, width): # функція приймає два параметри
     for i in range(height):
         print('*' * width)
```

Тепер наша функція `draw_box()` приймає два цілих параметри `height` - висота прямокутника і `width` - ширина прямокутника, і для її виклику нам потрібно обов'язково вказати їх.

Щоб вивести зірковий прямокутник розмірами 5 на 7, ми пишемо код:

```
draw_box(5, 7)
```

Результатом такого виклику функції `draw_box(5, 7)` буде:

```
*******
*******
*******
*******
*******
```

Щоб вивести прямокутник розмірами 10 на 15 ми пишемо код:

```
draw_box(10, 15)
```

Результатом такого виклику функції `draw_box(10, 15)` буде:

```
***************
***************
***************
***************
***************
***************
***************
***************
```

```
**************
**************
```

Тепер за допомогою нової версії функції `draw_box()` можемо в одній програмі виводити прямокутники різних розмірів. Наступний програмний код:

```
draw_box(3, 3)
print()
draw_box(5, 5)
print()
draw_box(4, 10)
```

виведе:

```
***
***
***

*****
*****
*****
*****
*****

**********
**********
**********
**********
```

На місце параметрів ми можемо підставляти не тільки цілочисленні константи, а й значення змінних. Наступний програмний код:

```
n = 3
m = 9
draw_box(n, m)
```

виведе:

```
*********
*********
*********
```

Ще приклади

Напишемо функцію `print_hello(n)`, яка приймає одне натуральне число *n* і друкує слово `Hello` рівно *n* разів.

```
def print_hello(n):
```

```
    print('Hello' * n)
```

Наступний програмний код:

```
print_hello(3)
print_hello(5)
times = 2
print_hello(times)
```

виведе:

```
HelloHelloHello
HelloHelloHelloHelloHello
HelloHello
```

Функцію `print_hello()` можна зробити гнучкішою, якщо передавати до неї ще один параметр – текст для виведення:

```
def print_text(txt, n):
    print(txt * n)
```

Наступний програмний код:

```
print_text('Hello', 5)
print_text('A', 10)
```

виведе:

```
HelloHelloHelloHelloHello
AAAAAAAAAA
```

Параметри VS аргументи

Аргумент – це будь-яка порція даних, що передається у функцію, коли функція викликається. ***Параметр*** – це змінна, яка отримує аргумент, переданий у функцію.

Для функції `draw_box(height, width):`

```
def draw_box(height, width):
    for i in range(height):
        print('*' * width)
```

параметрами є змінні `height` та `width`.

У момент виклику функції `draw_box(height, width):`

```
height = 10
draw_box(height, 9)
```

аргументами є `height` та `9`.

Параметри функцій часто називають ***параметричними змінними***.

Внесення змін до параметрів

Коли аргумент передається у функцію, параметрична змінна функції посилатиметься на значення цього аргументу. Однак будь-які зміни, які вносяться до параметричних змінних, не впливатимуть на аргумент.

Наступний програмний код:

```
def draw_box(height, width):
     height = 2
     width = 10
     for i in range(height):
         print('*' * width)

n = 5
m = 7
draw_box(n, m)
print(n, m)
```

виведе:

```
**********
**********
5 7
```

У тілі функції вносяться зміни в значення параметричних змінних `height` і `width`, проте це ніяк не вплинуло на значення змінних `n` і `m` з основної програми, які передавалися в якості аргументів в функцію `draw_box()`.

Примітка

Примітка 1. Функції в Python можуть приймати скільки завгодно параметрів.

Примітка 2. Іноді замість параметрів та аргументів говорять про ***формальні параметри та фактичні параметри***. Формальні параметри – змінні, які ми пишемо під час опису функції. Фактичні параметри – те, що реально підставляється під час виклику функції.

Вирішення задач 3-5 з практичної роботи 7

Локальні змінні

Локальними називаються змінні, оголошені всередині функції та доступні лише їй самій. Програмний код за межами функції доступу до них не має.

Розглянемо функцію `print_texas()`, яка виводить інформацію про кількість птахів, що мешкають у Техасі.

```
def print_texas():
     birds = 5000
     print('У Техасі мешкає', birds, 'птиц.')
```

У тілі функції ми створюємо змінну `birds`, якій надається значення, що дорівнює `5000`. Така змінна є локальною для функції `print_texas()`. Щоразу, коли змінної всередині функції присвоюється значення, ***створюється локальна змінна.*** Вона належить до функції, в якій створюється, і до неї отримує доступ лише програмний код цієї функції.

Термін ***"локальна"*** вказує на те, що змінна може використовуватися лише в цьому місці - усередині функції, в якій створюється.

Якщо програмний код однієї функції спробує звернутися до локальної змінної, що належить іншій функції, може статися помилка.

Розглянемо наступний програмний код:

```
def print_texas():
     birds = 5000
     print('У Техасі мешкає', birds, 'птиц.')

def print_california():
     print('У Каліфорнії мешкає', birds, 'птахів.')
```

Функція `print_california()` звертається до локальної змінної `birds` функції `print_texas()`. Виклик функції `print_california()` призводить до помилки:

```
NameError: name 'birds' is not defined
```

Локальні змінні приховані від інших функцій, тому інші функції можуть мати власні локальні змінні з тим самим ім'ям. Наприклад,

```
def print_texas():
     birds = 5000
```

```
    print('У Техасі мешкає', birds, 'птиц.')

def print_california():
    birds = 9000
    print('У Каліфорнії мешкає', birds, 'птахів.')
```

У кожній з цих двох функцій є локальна змінна під назвою `birds`. Але їх ніколи не видно одночасно, оскільки вони знаходяться у різних функціях.

Коли виконується функція `print_texas()`, видима змінна `birds`, значення якої дорівнює `5000`. Коли виконується функція `print_california()`, видима змінна `birds`, значення якої дорівнює `9000`.

Різні функції можуть мати локальні змінні з однаковими іменами, тому що вони не бачать локальних змінних один одного.

Область дії змінної

Область дії змінної – частина програми, у якій можна до неї звертатися, та функція, де вона створена. Змінна видима лише програмному коду у сфері її дії. Жодна інструкція за межами функції не може звертатися до такої змінної.

До локальної змінної не може звертатися програмний код, який з'являється всередині функції, ***перш ніж*** змінна була створена.

Наприклад, якщо в функції `print_texas()` поміняти місцями рядки коду:

```
def print_texas():
    print('У Техасі мешкає', birds, 'птиц.')
    birds = 5000
```

то при виклику цієї функції отримаємо помилку:

```
UnboundLocalError: local variable 'birds' referenced before assignment
```

Помилка виникла внаслідок передчасного звернення до ще не оголошеної локальної змінної `birds`.

Область дії параметричної змінної

Область дії параметричної змінної – функція, в якій цей параметр використовується. До параметричної змінної має доступ весь програмний код цієї функції.

Розглянемо вже відому нам функцію:

```
def draw_box(height, width):
     for i in range(height):
         print('*' * width)
```

Параметричні змінні тут `height, width`. Усередині функції оголошується одна локальна змінна `i`.

Примітки

Примітка 1. Параметрична змінна також локальна.

Примітка 2. Пам'ять для локальних змінних виділяється на час виконання цієї функції у спеціальній області, яка називається ***стеком.*** Після завершення роботи функції пам'ять звільняється, внутрішні результати функції не зберігаються від одного звернення до іншого.

Глобальні змінні

Глобальними називаються змінні, оголошені в основній програмі та доступні як програмі, так і всім її функціям.

Розглянемо наступний програмний код:

```
birds = 5000 # глобальна змінна

def print_texas():
     print('У Техасі мешкає', birds, 'птиц.')

def print_california():
     print('У Каліфорнії мешкає', birds, 'птахів.')
```

На початку програми створюємо глобальну змінну `birds`, значення якої дорівнює `5000`. Далі описуємо дві функції, що звертаються до глобальної змінної. Результатом виконання наступного коду:

```
print_texas()
print_california()
```

буде:

```
У Техасі мешкає 5000 птахів.
У Каліфорнії мешкає 5000 птахів.
```

Обмін інформацією між основною програмою та функціями здійснюється лише за допомогою параметрів функцій та глобальних змінних.

Функція може використовувати будь-які глобальні змінні крім тих, що мають ті ж імена, що і її локальні змінні. Якщо в функції оголошена локальна змінна з тим самим ім'ям, що в однієї з глобальних, то ця глобальна змінна стає недоступною в цій функції, і при вказівці ідентифікатора змінної відбудеться звернення до локальної змінної функції, а не до однойменної глобальної.

Розглянемо наступний програмний код:

```
birds = 5000 # глобальна змінна

def print_texas():
     birds = 1000 # локальна змінна
     print('У Техасі мешкає', birds, 'птиц.')

def print_california():
     birds = 7000 # локальна змінна
     print('У Каліфорнії мешкає', birds, 'птахів.')
```

На самому початку програми ми створюємо глобальну змінну `birds`, значення якої дорівнює `5000`. Далі ми описуємо дві функції, в яких створюються локальні змінні з таким самим ім'ям `birds`. Таким чином, при зверненні до змінної `birds` всередині функцій відбуватиметься звернення саме до локальної змінної.

Результатом виконання наступного коду:

```
print_texas()
print_california()
```

буде:

```
У Техасі мешкає 1000 птахів.
У Каліфорнії мешкає 7000 птахів.
```

Більшість програмістів згодні, що слід обмежити використання глобальних змінних або не використовувати їх взагалі. Причини такі.

- ***Глобальні змінні ускладнюють налагодження програми***. Значення глобальної змінної можна змінити будь-якою інструкцією в програмному файлі. Якщо виявиться, що в глобальній змінній зберігається неправильне значення, то доведеться знайти всі інструкції, які до неї звертаються, щоб визначити, звідки надходить погане значення. У програмі із тисячами рядків коду зробити це непросто.
- ***Функції, які використовують глобальні змінні, зазвичай залежить від цих змінних.*** Якщо виникне потреба застосувати таку

функцію в іншій програмі, швидше за все, доведеться цю функцію перепроектувати, щоб вона не спиралася на глобальну змінну.

- ***Глобальні змінні ускладнюють розуміння програми***. Глобальна змінна може бути модифікована будь-якою інструкцією у програмі. При необхідності розібратися в якійсь частині програми, яка використовує глобальну змінну, доведеться дізнатися про всі інші частини програми, які звертаються до цієї глобальної змінної.

У більшості випадків слід створювати змінні локально і передавати їх в якості аргументів у функції, яким потрібно до них звернутися.

Глобальні константи

Хоча слід уникати використання глобальних змінних, у програмі допускається застосування глобальних констант. Глобальна константа – глобальне ім'я, що посилається на постійне значення. Оскільки значення глобальної константи не може бути змінено під час виконання програми, можна не турбуватися про потенційні небезпеки, які зазвичай пов'язані з використанням глобальних змінних.

Незважаючи на те, що мова Python не дозволяє створювати реальні глобальні константи, їх можна імітувати за допомогою глобальних змінних. Якщо глобальна змінна не оголошується з використанням ключового слова `global` усередині функції, то всередині цієї функції ви не зможете змінити значення цієї глобальної змінної.

Математичний модуль `math` визначає дві глобальні змінні, `math.pi` та `math.e`, яким присвоєно математичні значення констант π = `3.14159265` та e = `2.71828`.

Ключове слово global

Якщо потрібно, щоб інструкція всередині функції надавала значення глобальної змінної, потрібний додатковий крок. У цьому випадку глобальна змінна повинна бути оголошена всередині функції.

Розглянемо наступний програмний код:

```
def print_texas():
     global birds
     birds = 5000
     print('У Техасі мешкає', birds, 'птиц.')

def print_california():
     print('У Каліфорнії мешкає', birds, 'птахів.')
```

```
print_texas()
print_california()
```

Результатом виконання наступного коду:

```
print_texas()
print_california()
```

буде:

```
У Техасі мешкає 5000 птахів.
У Каліфорнії мешкає 5000 птахів.
```

Функція із поверненням значення

Функція із поверненням значення схожа на функцію без повернення значення тим, що:

- це набір інструкцій, що виконує певну задачу;
- коли необхідно виконати функцію, її викликають.

Однак, коли функція з поверненням значення завершується, вона повертає значення в частину програми, яка її викликала. Значення, що повертається з функції, використовується як будь-яке інше: воно може бути присвоєно змінній, виведено на екран, використано в математичному виразі (якщо це число) і т. д.

Функція із поверненням значення повертає значення назад у ту частину програми, яка її викликала.

Ми вже стикалися з багатьма функціями із поверненням значень:

- функція `int()` – перетворює рядок до цілого числа та повертає його;
- функція `float()` – перетворює рядок до дійсного числа і повертає його;
- функція `range()` – повертає послідовність цілих чисел 0, 1, 2, …;
- функція `abs()` - повертає абсолютне значення числа (модуль числа);
- функція `len()` – повертає довжину рядка або списку.

Функцію з поверненням значення пишуть так само, як і без, але вона повинна мати інструкцію `return`.

Ось загальний формат визначення функції з поверненням значення в Python:

```
def назва_функції():
     блок коду
     return вираз
```

В функції має бути інструкція `return`, що набуває форми:

```
return вираз
```

Значення виразу, яке слідує за ключовим словом `return`, буде відправлено до частини програми, яка викликала функцію. Це може бути змінна або вираз, наприклад, математичний.

При вивченні дійсних чисел ми вирішували задачу про переведення градусів за шкалою Фаренгейта в градуси за шкалою Цельсія за формулою $C = \frac{5}{9}(F - 32)$.

Напишемо функцію, яка здійснює переведення:

```
def convert_to_celsius(temp):
     result = (5/9) * (temp - 32)
     return result
```

Задача цієї функції – прийняти одне число `temp` в якості аргумента – кількість градусів за шкалою Фаренгейта, і повернути інше – кількість градусів за шкалою Цельсія.

Розглянемо її роботу. Перша інструкція в блоці функції надає значення `(5/9) * (temp - 32)` змінній `result`. Потім виконується інструкція `return`, яка призводить до завершення виконання функції та відправляє значення із змінної `result`, назад у ту частину програми, яка викликала цю функцію.

```
# функція переведення градусів Фаренгейта на градуси Цельсія
def convert_to_celsius(temp):
     result = (5/9) * (temp - 32)
     return result

# основна програма
temp = float(input('Введіть кількість градусів за Фаренгейтом: '))
celsius = convert_to_celsius(temp)
print(celsius) # градуси Цельсія
```

Основна програма отримує від користувача одне число – значення у градусах Фаренгейта, і викликає функцію, передаючи значення змінної `temp` як

аргумент. Значення, яке повертається з функції `convert_to_celsius`, надається змінній `celsius`.

Використання інструкції return по максимуму

Погляньмо ще раз на функцію `convert_to_celsius()`:

```
def convert_to_celsius(temp):
     result = (5/9) * (temp - 32)
     return result
```

Зверніть увагу, що всередині цієї функції відбуваються дві речі: по-перше, змінній `result` присвоюється значення виразу `(5/9)* (temp – 32)`, і по-друге, значення змінної `result` повертається з функції. Ця функція добре справляється з поставленою перед нею задачею, але її можна спростити. Оскільки інструкція `return` повертає значення виразу, змінну `result` усуваємо та переписуємо функцію так:

```
def convert_to_celsius(temp):
     return (5/9) * (temp - 32)
```

Ця версія функції не зберігає значення `(5/9) * (temp - 32)` в окремій змінній, а одразу повертає значення виразу за допомогою інструкції `return`. Робить те саме, що й попередня версія, але за один крок.

Використання декількох return

В одній функції може бути скільки завгодно інструкцій `return`. Розглянемо функцію `convert_grade()`, яка переводить стобальну оцінку до п'ятибальної:

```
def convert_grade(grade):
     if grade >= 90:
         return 5
     elif grade >= 80:
         return 4
     elif grade >= 70:
         return 3
     elif grade >= 60:
         return 2
     else:
```

```
        return 1

# основна програма
grade = int(input('Введіть вашу позначку за 100-бальною системою: '))
print(convert_grade(grade))
```

Функція `convert_grade()` використовує 5 інструкцій `return`. Кожна з них повертає відповідне значення та завершує роботу функції.

Функцію `convert_grade()` можна переписати за допомогою однієї інструкції `return`:

```
def convert_grade(grade):
    if grade >= 90:
        result = 5
    elif grade >= 80:
        result = 4
    elif grade >= 70:
        result = 3
    elif grade >= 60:
        result = 2
    else:
        result = 1

    return result
```

Примітки

Примітка 1. Функції із поверненням значення надають ті ж переваги, що функції без повернення значення:

- спрощують програмний код;
- зменшують дублювання коду;
- спрощують тестування коду;
- збільшують швидкість розробки;
- сприяють роботі у команді.

Примітка 2. Графічна інтерпретація роботи функції із поверненням значення:

Примітка 3. `result` – гарна назва для змінної, значення якої повертається із функції.

Рішення задач

Задача 1. Напишіть функцію, яка повертає довжину гіпотенузи прямокутного трикутника за відомими значеннями його катетів.

Рішення. Для знаходження довжини гіпотенузи нам потрібно застосувати теорему Піфагора: квадрат гіпотенузи прямокутного трикутника, дорівнює сумі квадратів його катетів. Іншими словами, якщо a, b – довжини катетів, а c – довжина гіпотенузи, має місце рівність:

$$c^2 = a^2 + b^2 \Rightarrow c = \sqrt{a^2 + b^2}$$

Функція, що обчислює довжину гіпотенузи, може мати вигляд:

```
def compute_hypotenuse(a, b):
     c = (a**2 + b**2)**0.5
     return c
```

Для отримання квадратного кореня ми використовували оператор зведення в ступінь. Нагадаємо, що результатом обох виразів: `math.sqrt(c)` та `c**0.5` є одне число $\sqrt{c}$.

Наступний програмний код:

```
print(compute_hypotenuse(3, 4))
print(compute_hypotenuse(5, 12))
print(compute_hypotenuse(1, 1))
```

виведе:

```
5.0 # довжина гіпотенузи трикутника з катетами 3 та 4
13.0 # довжина гіпотенузи трикутника з катетами 5 та 12
1.4142135623730951 # довжина гіпотенузи трикутника з катетами 1 і 1
```

Якщо потрібно передати програмі числа, введені з клавіатури, ми пишемо наступний код:

```
x = int(input())
y = int(input())

hypotenuse = compute_hypotenuse(x, y)

print(hypotenuse)
```

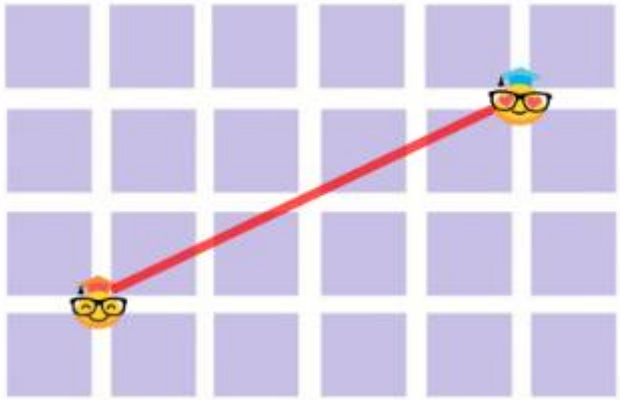
У модулі `math` є вбудована функція `hypot(x, y)`, яка повертає довжину гіпотенузи прямокутного трикутника з катетами `x` і `y`.

Однією з основних переваг функцій є можливість їх повторного використання для вирішення схожих задач. Розглянемо задачу знаходження відстані між двома точками.

Задача 2. Напишіть функцію `get_distance(x1, y1, x2, y2)`, яка обчислює відстань між точками $(x_1; y_1)$ та $(x_2; y_2)$.

Рішення. Відстань між двома точками $(x_1; y_1)$ та $(x_2; y_2)$ визначається за формулою

$$\rho = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$



Неважко помітити, що відстань, що шукається, - це довжина гіпотенузи прямокутного трикутника з катетами рівними $|x_1 - x_2|$ та $|y_1 - y_2|$.

Функція, що обчислює відстань між точками, може мати вигляд:

```
def get_distance(x1, y1, x2, y2):
     return compute_hypotenuse(x1 - x2, y1 - y2)
```

Для підрахунку шуканої відстані ми використовуємо вже створену нами функцію `compute_hypotenuse` передаючи їй в якості аргументів числа `x1 - x2` і `y1 - y2`.

Основна програма має вигляд:

```
x1, y1 = float(input()), float(input()) # зчитуємо координати першої точки
x2, y2 = float(input()), float(input()) # зчитуємо координати другої точки

print(get_distance(x1, y1, x2, y2)) # обчислюємо та виводимо відстань між точками
```

Задача 3. Напишіть функцію `sum_digits(n)`, яка приймає в якості аргументу натуральне число і повертає суму його цифр.

Рішення. Функція `sum_digits(n)` може мати вигляд:

```
def sum_digits(n):
     result = 0
     while n > 0:
         result += n % 10
         n //= 10
     return result
```

Основна програма має вигляд:

```
n = int(input())
print(sum_digits(n)) # обчислюємо та виводимо суму цифр ліченого числа
```

Задача 4. Напишіть функцію `compute_average(numbers)`, яка приймає в якості аргументу список чисел і повертає середнє значення елементів списку.

Рішення. Для підрахунку середнього значення елементів списку потрібно обчислити суму всіх елементів та їх кількість, тобто використовувати функції `sum()` та `len()`. Функція `compute_average(numbers)` може мати вигляд:

```
def compute_average(numbers):
     return sum(numbers) / len(numbers)
```

Основна програма має вигляд:

```
numbers = [1, 3, 5, 1, 6, 8, 10, 2]
print(compute_average(numbers)) # обчислюємо та виводимо середнє значення елементів списку
```

Результатом роботи такої програми буде число 4.5, яке є середнім значенням.

Вирішення задач 6-11 з практичної роботи 7

Злиття двох відсортованих списків

Злиття двох відсортованих списків в один — важлива задача інформатики. Вона природно виникає при сортуванні списків з використанням сортування злиттям.

Нехай дані два відсортованих за зростанням списки чисел `list1` і `list2`:

```
list1 = [3, 10, 11, 12, 47, 57, 58, 63, 77, 79, 80, 95]
list2 = [0, 11, 12, 20, 24, 26, 47, 48, 53, 65, 70, 81, 84, 84, 90]
```

Найпростіше рішення задачі злиття списків використовує метод списку `sort()`:

```
def merge(list1, list2):
     result = list1 + list2 # створюємо результуючий список
     result.sort() # сортуємо список вбудованим методом sort()
     return result # повертаємо відсортований список


list1 = [3, 10, 11, 12, 47, 57, 58, 63, 77, 79, 80, 95]
list2 = [0, 11, 12, 20, 24, 26, 47, 48, 53, 65, 70, 81, 84, 84, 90]
list3 = merge(list1, list2) # викликаємо функцію злиття двох відсортованих списків

print(list3)
```

Результатом роботи такого коду буде список:

```
[0, 3, 10, 11, 11, 12, 12, 20, 24, 26, 47, 47, 48, 53, 57, 58, 63, 65, 70, 77, 79, 80, 81, 84, 84 , 90, 95]
```

І хоча функція `merge()` повністю справляється зі своїм завданням, вона абсолютно не враховує те, що два списки `list1` та `list2` вже відсортовані.

Швидке злиття двох відсортованих списків в один

Нехай ми маємо два вже відсортовані за зростанням списки `list1` та `list2`.

Алгоритм швидкого злиття наступний:

1. Створюємо чисельні покажчики `p1 = 0` і `p2 = 0` на початку обох списків `list1` і `list2` відповідно;
2. На кожному кроці беремо менший із двох елементів `list1[p1]` і `list2[p2]`;
3. Записуємо його до результуючого списку;
4. Збільшуємо на 1 покажчик на перший елемент списку (`p1` або `p2`), з якого було взято елемент;
5. Коли один із початкових списків закінчився, додаємо всі елементи другого списку, що залишилися, в результуючий список.

```
def quick_merge(list1, list2):
     result = []

     p1 = 0 # покажчик першого елементу списку list1
```

```
    p2 = 0 # покажчик першого елементу списку list2

    while p1 < len(list1) and p2 < len(list2): # поки не закінчився
хоча б один список
        if list1[p1] <= list2[p2]:
            result.append(list1[p1])
            p1 += 1
        else:
            result.append(list2[p2])
            p2 += 1

    if p1 < len(list1): # причеплення залишку
        result += list1[p1:]
    else: # інакше причіплюємо залишок іншого списку
        result += list2[p2:]

    return result
```

Наступний програмний код:

```
list1 = [3, 10, 11, 12, 47, 57, 58, 63, 77, 79, 80, 95]
list2 = [0, 11, 12, 20, 24, 26, 47, 48, 53, 65, 70, 81, 84, 84, 90]
list3 = quick_merge(list1, list2)
print(list3)
```

виведе:

```
[0, 3, 10, 11, 11, 12, 12, 20, 24, 26, 47, 47, 48, 53, 57, 58, 63, 65,
70, 77, 79, 80, 81, 84, 84 , 90, 95]
```

Вирішення задачі 12 з практичної роботи 7

Повернення булевих значень

Python дозволяє писати булеві функції, що повертають або істину (True), або кривду (False). Бульову функцію можна застосовувати для перевірки умови, тоді значення True і False сигналізуватимуть про її виконання.

Булеві функції широко застосовуються для спрощення складних умов, що перевіряються у структурах прийняття рішення та структурах з повторенням. Наприклад, напишемо програму, яка просить користувача ввести число, та визначає парне воно чи непарне.

Це можна зробити так:

```
number = int(input())
if number % 2 == 0:
```

```
     print('Це число парне.')
else:
     print('Це число непарне.')
```

Цей фрагмент коду буде легше зрозуміти, якщо написати булеву функцію `is_even()`, яка приймає число як аргумент і повертає `True`, якщо воно парне, і `False` якщо непарне.

```
def is_even(number):
     if number % 2 == 0:
         return True
     else:
         return False
```

Тепер можна переписати інструкцію `if-else` основної програми так, щоб вона для визначення парності змінної `number` викликала функцію `is_even()`:

```
number = int(input())
if is_even(number):
     print('Це число парне.')
else:
     print('Це число непарне.')
```

Так логіку програми легше зрозуміти, а функцію можна викликати в програмі щоразу, коли необхідно перевірити парність числа.

Використання булевих функцій для валідації вхідних даних

Булеві функції також можна використовувати для ***спрощення складного коду валідації вхідних даних***. Наприклад, у програмі, яка пропонує користувачеві ввести номер моделі виробу, де можливі лише значення 100, 200 і 300, можемо написати такий код:

```
model = int(input())

while model != 100 and model != 200 and model != 300:
     print('Допустимими номерами моделей є 100, 200 і 300.')
     model = int(input())
```

Цикл валідації використовує довгий складовий булевий вираз, і повторюється до тих пір, поки `model` не дорівнюватиме 100 і 200 або 300.

Разом про те, цикл валідації можна спростити, написавши булеву функцію перевірки змінної `model`, і викликаючи їх у циклі. Напишемо функцію `is_invalid()`, яка приймає один параметр `model` і повертає значення `True`, якщо модель неприпустима і `False` інакше. Тоді цикл валідації можна переписати так:

```
while is_invalid(model):
     print('Допустимими номерами моделей є 100, 200 і 300.')
     model = int(input())
```

Після цієї зміни цикл стає легше читати. Тепер цілком очевидно, що цикл повторюється доти, доки номер моделі неприпустимий. Нижче наведений фрагмент коду показує, як можна було б написати функцію `is_invalid()`. Вона приймає номер моделі як аргумент, і якщо аргумент не дорівнює 100, 200 та 300, то ця функція повертає `True`, говорячи, що він неприпустимий. В іншому випадку функція повертає `False`.

```
def is_invalid(model):
     if model != 100 and model != 200 and model != 300:
         return True
     else:
         return False
```

Створення функцій, що реалізують таку просту логіку — не завжди оптимальне рішення, тому що збільшує розмір коду і веде до витрат часу на виклик функції і повернення результату, що може позначитися на продуктивності програми.

Вирішення задач 13-21 з практичної роботи 7

Функції із поверненням кількох значень

В Python функції не обмежені поверненням лише одного значення. Після інструкції `return` можна визначити багато виразів, розділених комами:

```
return вираз 1, вираз 2, вираз 3 ...
```

Наступний програмний код визначає функцію `get_powers(num)`, яка приймає як аргумент число `num` і повертає його квадрат, куб і четвертий ступінь.

```
def get_powers(num):
     return num**2, num**3, num**4
```

Результатом виконання наступного коду:

```
a, b, c = get_powers(2)
print(a)
print(b)
print(c)
```

буде:

```
4
8
16
```

Функції, що повертають кілька значень, – виняткова особливість мови Python. У більшості мов програмування для повернення кількох значень використовують результуючий тип даних – список, який може містити кілька значень.

Розглянемо ще один приклад. Нехай потрібно написати функцію, яка знаходить точку перетину двох ***непаралельних*** прямих $ax + by = e$ та $cx + dy = f$. Тобто потрібно вирішити систему рівнянь:

$$\begin{cases} ax + by = e \\ cx + dy = f \end{cases}$$

Неважко знайти рішення цієї системи:

$$\begin{cases} ax + by = e \\ cx + dy = f \end{cases} \Leftrightarrow \begin{cases} x = \dfrac{d \cdot e - b \cdot f}{a \cdot d - b \cdot c} \\ y = \dfrac{a \cdot f - c \cdot e}{a \cdot d - b \cdot c} \end{cases}$$

Програмний код, який вирішує задачу, має вигляд:

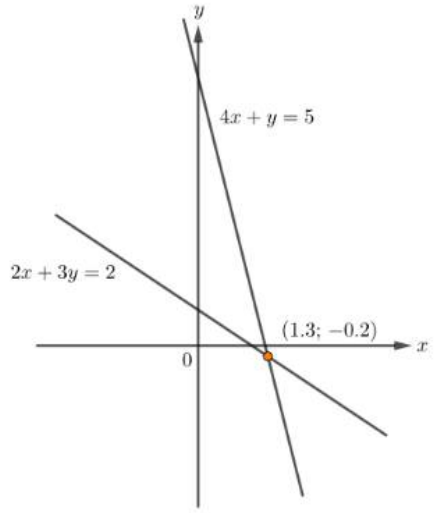
```
def solve(a, b, c, d, e, f):
     x = (d * e - b * f) / (a * d - b * c)
     y = (a * f - c * e) / (a * d - b * c)
     return x, y
```

Наступний код:

```
xsol, ysol = solve(2, 3, 4, 1, 2, 5)
print('Рішенням системи є числа', 'x =', xsol, 'y =', ysol)
```

виведе

```
Рішенням системи є числа x = 1.3 y = -0.2
```



Вирішення задач 22-24 з практичної роботи 7

Переваги використання функцій

Розбиваючи програми на функції отримуємо:

- **Простіший код**. Розбитий на функції код програми простіше та легше для розуміння. Декілька невеликих функцій набагато легше читати, ніж одну довгу послідовність інструкцій;
- **Повторне використання коду**. Функції дозволяють уникнути багаторазового повторення коду у програмі. Якщо якась операція в програмі виконується в декількох місцях, то можна один раз написати для неї функцію і виконувати її, коли знадобиться.
- **Простіше тестування**. Коли кожна задача у програмі міститься у власній функції, програмісти можуть індивідуально протестувати кожну функцію і визначити, чи вона виконує своє завдання правильно.
- **Швидша розробка**. Припустимо, що програміст чи команда програмістів розробляє багато програм. Вони виявляють загальні задачі у різних програм, наприклад з'ясування імені користувача і пароля, виведення поточного часу. Щоразу писати програмний код цих задач немає сенсу. Для завдань, які часто зустрічаються, пишуть функції, і включають до складу будь-якої програми, що їх потребує.
- **Спрощення командної роботи**. Коли програма розробляється як набір функцій, різним програмістам можна доручити написання окремих функцій.

Що виділяти у функції?

У функцію можна виділити будь-який закінчений фрагмент програми. Можна орієнтуватися на рекомендації:

- Коли кілька разів пишете в програмі одну і ту ж послідовність команд, необхідність введення функції набуває характеру гострої внутрішньої потреби;
- Іноді розмаїтість дрібниць заступає головне і корисно прибрати в функцію подробиці, що приховують сенс основної програми;
- Корисно розбити довгу програму на складові, як книгу розбивають на глави, при цьому основна програма стає схожою на зміст;
- Складні приватні алгоритми корисно налагодити окремо в невеликих тестуючих програмах. Включити їх до основної програми буде легко, якщо вони оформлені у вигляді функцій. Наприклад, функції сортувань;
- Зроблене добре в одній програмі, хочеться перенести на нові. Для повторного використання краще відразу виділяти у програмі корисні алгоритми в окремі функції, а функції збирати пакети.

ЛІТЕРАТУРА

1. Розум М.В. Методичні вказівки щодо проведення занять з дисципліни «Основи програмування на мові Python» / М.В. Розум. – Одеса: ОНМУ, 2023. – 159 с.
2. Олександр Мізюк. Путівник мовою програмування Python. [Електронний ресурс]. Режим доступу: https://pythonguide.rozh2sch.org.ua/
3. Python Підручник [Електронний ресурс]: https://w3schoolsua.github.io/python/index.html#gsc.tab=0
4. Підручник мови Python [Електронний ресурс]: http://docs.linux.org.ua/Програмування/Python/Підручник_мови_Python/
5. Освітня платформа України Prometheus. Основи Python. [Електронний ресурс]: https://prometheus.org.ua/prometheus-plus/python-beetroot-course/
6. Освітня платформа України Prometheus. Основи програмування CS50 2019. [Електронний ресурс]: https: /prometheus.org.ua course/course-v1:Prometheus+CS50+2019_T1
7. Освітня платформа України Prometheus. Вебпрограмування з Python та JavaScript CS50. [Електронний ресурс]: https: //prometheus.org.ua /course/course-v1:Prometheus+CS50+2021_T1
8. Освітня платформа України Prometheus. CS50: Основи програмування для бізнес-професіоналів. [Електронний ресурс]: https: /prometheus.org.ua course/course-v1:Prometheus+CS50S101+2023_T1
9. Освітня платформа Stepik [Електронний ресурс]: https://stepik.org
10. Rozum M.V., Gavrilyuk A.V. Development of software for the implementation of the analytic hierarchy method // Інформаційні управляючі системи та (ІУСТ-ОДЕСА-2013). Мат-ли міжнар. наук.-практ. конф., 8-10 жовтня 2013 г., Одеса. – Одеса: «ВидавІнформ» ОНМА, 2013. С.57-59.
11. Розум М.В. Розробка програмного забезпечення для кластерного аналізу даних // 71 проф.-виклад. науково-технічна конф. (29-31 травня 2018р.): Зб. тез допов. – Одеса, ОНМУ, 2018. – С.126-127.
12. Розум М.В. Розробка ВЕБ-інтерфейсу інформаційної системи виявлення збіжності текстових даних для цифрових документів // Проєктний та логістичний менеджмент: нові знання на базі двох методологій. Матеріали II наук.-практ. конференції, 10 листопада 2022р. Том 6: зб. Наукових праць.:– Одеса: КУПРІЄНКО СВ, 2022. – С. 157-158.
13. Розум М.В., Рудачевський Д. Розробка програми відображення розкладу занять // Інформатика, інформаційні системи та технології:

тези допов. 20 всеукр. конф. студентів і молодих науковців. Одеса, 28.04.2023 р. – Одеса, 2023. – С.131-134.

14. Розум М.В., Ігнатьєва К.М. Реалізація методу структурування альтернатив за допомогою мови Python // Матеріали XI Всеукраїнської науково-практичної конференції студентів та молодих вчених «Проблеми і перспективи розвитку транспорту», 21 квітня 2023 року. – частина 2. – 2023. – С. 34–36. https://www.onmu.odessa.ua/ua/konf-2/3379-xi-conf-tr-2023.htm