

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ  
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
ім. ІГОРЯ СІКОРСЬКОГО"

**А.В. Яковенко**

# **Основи програмування. Python. Частина 1**

*Затверджено Вченою радою НТУУ "КПІ ім. Ігоря Сікорського"  
як підручник для студентів які навчаються за спеціальністю  
122 "Комп'ютерні науки"  
спеціалізацією "Інформаційні технології в біології та медицині"*

Київ  
КПІ ім. Ігоря Сікорського  
2018

УДК

**Рецензенти:** *Подчашинський Юрій Олександрович*, д.т.н., професор  
*Великоіваненко Олена Андріївна*, канд. фіз.-мат. наук, ст.н.с.  
**Відповідальний редактор:** *Добровська Людмила Миколаївна*, канд. пед. наук, доцент

**Гриф надано Вченою радою КПІ ім. Ігоря Сікорського (протокол № 9 від 01.10.2018 р.)**

Електронне мережне навчальне видання  
*Яковенко Альона Вікторівна*, канд. техн. наук  
Основи програмування. Python. Частина 1

**Основи програмування. Python. Частина 1** [Електронний ресурс]: підручник для студ. спеціальності 122 "Комп'ютерні науки", спеціалізації "Інформаційні технології в біології та медицині" / А. В. Яковенко ; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 1,59 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2018. – 195 с.

У підручнику наведено базові поняття та означення основ обчислювальної техніки, апаратного та програмного забезпечення комп'ютерів, арифметичних основ ЕОМ та основ програмування мовою Python. Представлено опис теоретичних засад та вирішення практичних задач до вивчення дисципліни "Основи програмування", зокрема основних алгоритмічних структур, синтаксису мови Python та основ функціонального і модульного програмування. Матеріал супроводжується великою кількістю прикладів.

Для студентів, аспірантів та викладачів вищих навчальних закладів з напрямку інформаційні технології.

© А. В. Яковенко, 2018  
© КПІ ім. Ігоря Сікорського, 2018

## ЗМІСТ

<b>ВСТУП .....</b>	<b>6</b>
<b>РОЗДІЛ 1. ТЕХНОЛОГІЯ РОЗРОБКИ КОМП'ЮТЕРНИХ ПРОГРАМ .....</b>	<b>9</b>
<b>1.1. Архітектура цифрового комп'ютера .....</b>	<b>9</b>
<b>1.2. Подання даних.....</b>	<b>12</b>
<b>1.3. Поняття алгоритму. Властивості алгоритмів .....</b>	<b>24</b>
<b>1.4. Машинна програма.....</b>	<b>31</b>
<b>1.5. Мови програмування.....</b>	<b>33</b>
<b>1.6. Особливості мови програмування Python.....</b>	<b>37</b>
<b>Завдання на комп'ютерний практикум.....</b>	<b>39</b>
<b>Запитання для самоконтролю.....</b>	<b>39</b>
<b>РОЗДІЛ 2. БАЗОВІ ПОНЯТТЯ МОВИ PYTHON.....</b>	<b>41</b>
<b>2.1. Базовий синтаксис.....</b>	<b>41</b>
<b>2.2. Лексеми та ідентифікатори .....</b>	<b>43</b>
<b>2.3. Змінні.....</b>	<b>47</b>
<b>2.4. Типи даних .....</b>	<b>48</b>
<b>2.5. Прості типи даних. Числа .....</b>	<b>50</b>
<b>2.6. Прості логічні вирази та логічний тип даних.....</b>	<b>58</b>
<b>2.7. Логічні оператори.....</b>	<b>60</b>
<b>2.8. Складні структури даних. Рядки .....</b>	<b>64</b>

<b>2.9. Складні структури даних. Списки.....</b>	<b>83</b>
<b>2.10. Складні структури даних. Кортежі.....</b>	<b>94</b>
<b>2.11. Складні структури даних. Словники .....</b>	<b>97</b>
<b>Завдання на комп'ютерний практикум.....</b>	<b>104</b>
<b>Запитання для самоконтролю.....</b>	<b>105</b>
<b>РОЗДІЛ 3. АЛГОРИТМІЧНІ СТРУКТУРИ В МОВІ PYTHON</b>	<b>106</b>
<b>3.1. Основні алгоритмічні структури .....</b>	<b>106</b>
<b>3.2. Реалізація алгоритмів з розгалуженням.....</b>	<b>108</b>
<b>3.3. Реалізація циклічних алгоритмів .....</b>	<b>114</b>
<b>Завдання на комп'ютерний практикум.....</b>	<b>129</b>
<b>Запитання для самоконтролю.....</b>	<b>130</b>
<b>РОЗДІЛ 4. ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ .....</b>	<b>131</b>
<b>4.1. Функції.....</b>	<b>131</b>
<b>4.2. Рекурсія.....</b>	<b>146</b>
<b>4.3. Модульність в Python .....</b>	<b>150</b>
<b>Завдання на комп'ютерний практикум.....</b>	<b>166</b>
<b>Запитання для самоконтролю.....</b>	<b>166</b>
<b>РОЗДІЛ 5. РОБОТА З ФАЙЛАМИ.....</b>	<b>168</b>
<b>5.1. Уведення інформації у файли .....</b>	<b>168</b>
<b>5.2. Зчитування даних з файлу .....</b>	<b>170</b>
<b>Завдання на комп'ютерний практикум.....</b>	<b>173</b>

<b>Питання для самоконтролю .....</b>	<b>173</b>
<b>РОЗДІЛ 6. ВИНЯТКИ.....</b>	<b>174</b>
<b>6.1. Загальні поняття.....</b>	<b>174</b>
<b>6.2. Оброблення винятків.....</b>	<b>175</b>
<b>6.3. Класи вбудованих винятків .....</b>	<b>179</b>
<b>Запитання для самоконтролю.....</b>	<b>180</b>
<b>Завдання до розрахункової роботи .....</b>	<b>181</b>
<b>ПРЕДМЕТНИЙ ПОКАЖЧИК.....</b>	<b>182</b>
<b>СПИСОК ЛІТЕРАТУРИ.....</b>	<b>185</b>
<b>Додаток 1. Функції та методи рядків.....</b>	<b>187</b>
<b>Додаток 2. Методи списків .....</b>	<b>193</b>
<b>Додаток 3. Методи словників.....</b>	<b>194</b>

## ВСТУП

Навчальна дисципліна "Основи програмування" належить до циклу професійної підготовки навчального плану підготовки бакалавра циклу навчальних дисциплін професійної та практичної підготовки (за вибором студентів).

**Предмет навчальної дисципліни** – процес навчання і підготовки зі спеціальності 122 "Комп'ютерні науки" за спеціалізацією "Інформаційні технології в біології та медицині" першого (бакалаврського) рівня вищої освіти ступеня бакалавра, який дозволить використовувати знання основ програмування при розробці інформаційних технологій, що зараз охоплюють майже всі сфери життя і діяльності та відіграють важливу роль в біомедичних застосуваннях.

**Для дисципліни визначені такі міждисциплінарні зв'язки:**

У структурно-логічній схемі програми підготовки фахівця дисципліна забезпечує такі навчальні дисципліни та кредитні модулі:

- "Проектування та аналіз обчислювальних алгоритмів" (5/I);
- "Алгоритмізація та програмування" (1/II);
- "Об'єктно-орієнтоване програмування" (2/II).

**Метою навчальної дисципліни** є формування у студентів **здатностей:**

- застосовувати професійні знання й уміння на практиці СК-2;
- адаптуватися до різних професійних ситуацій, проявляти творчий підхід, ініціативу СК-3;
- аналізувати проблеми, ставити постановку цілей і завдань, виконувати вибір способу й методів дослідження, а також оцінку його якості СК-5;
- вирішувати проблеми в професійній діяльності на основі аналізу й синтезу ІК-2;

– застосовувати сучасні парадигми програмування під час програмної реалізації професійних задач ПК-11.

Розвиток сучасних технологій неможливий без використання комп'ютерної техніки та програмного забезпечення. Підготовка фахівців у галузі інформаційних технологій в біології та медицині вимагає великих знань та навичок володіння комп'ютерною технікою, а також знання основ алгоритмізації та програмування мовами високого рівня.

Підручник призначений для вивчення основ обчислювальної техніки та програмування високорівневою мовою Python для студентів першого курсу які навчаються за спеціальністю 122 "Комп'ютерні науки". Мова Python обрана через легкість вивчення та оскільки починаючи з другого курсу студенти поступово вивчають мови C++, Java та ін.

Python – це універсальна мова, що широко використовується в усьому світі для самих різних цілей – бази даних і оброблення текстів, вбудовування інтерпретатора в ігри, програмування GUI, швидке створення прототипів (RAD) програмування Internet і Web додатків – серверних (CGI), клієнтських (роботи), Web-серверів і серверів додатків.

Серед переваг мови Python можна виділити переносимість написаних програм, на комп'ютери різної архітектури та з різними операційними системами, лаконічність запису алгоритмів, можливість отримати ефективний код програм за швидкістю виконання. Зручність мови Python основане на тому, що вона є мовою високого рівня, має набір конструкцій структурного програмування та підтримує модульність. Гнучкість та універсальність мови Python забезпечує її широке розповсюдження.

Навчальний посібник орієнтований на програмістів початківців, що мають початкові поняття про основи алгоритмізації. Кожен розділ посібника містить відповідні приклади, що представляють собою програми, що написані

мовою Python. Приклади допоможуть ефективно засвоїти основи програмування мовою Python.

**У першому** розділі підручника розглянуто архітектуру та структуру комп'ютерів, основні відомості про апаратне та програмне забезпечення, властивості алгоритмів, приділено увагу графічному способу подання алгоритмів, наведено умовні графічні позначення у схемах алгоритмів, правила графічного запису алгоритмів та базові структури алгоритмів. Крім цього, наведено математичні основи ЕОМ, форми та формати подання даних, основні системи числення, правила та приклади переведення як цілих так і дробових чисел з однієї системи числення в іншу.

**У другому** розділі розглядаються основні засоби програмування в мові Python. Особливу увагу приділено простим та складним типам даних.

**У третьому** розділі розглянуто основні алгоритмічні конструкції (умови, цикл).

**Четвертий** розділ присвячено елементам функціонального та модульного програмування. Визначено поняття рекурсії та наведено детальні приклади роботи рекурсивних функцій.

**У п'ятому** розділі описано роботу з файлами і каталогами.

В заключному **шостому** розділі описано можливості оброблення виняткових ситуацій.

Автор висловлює велику вдячність рецензентам – д-ру. техн. наук, професору Ю. О. Подчашинському і канд. фіз.-мат. наук, ст.н.с. О. А. Великоіваненко за цінні поради, які сприяли покращенню підручника.



# РОЗДІЛ 1. ТЕХНОЛОГІЯ РОЗРОБКИ КОМП'ЮТЕРНИХ ПРОГРАМ

## 1.1. Архітектура цифрового комп'ютера

*ЕОМ* – *електронна обчислювальна машина*, що здатна виконувати обчислювальні задачі будь-якої складності за наявності відповідної програми.

Усе забезпечення комп'ютера ділиться на дві частини: *апаратну* та *програмну*.

До апаратного забезпечення відносяться пристрої і прилади, що утворюють апаратну конфігурацію. Сучасні комп'ютери й обчислювальні комплекси мають *блоково-модульну конструкцію*. Основний блок персонального комп'ютера (ПК) – системний блок. Він містить блок електроживлення, кріпильні елементи для материнської (системної) плати, електронних плат і дисководів [2].

*Архітектурою комп'ютера* називають склад і взаємозв'язок основних пристроїв комп'ютера. Найбільш поширеною є архітектура зображена на рис 1.1, що відповідає принципам, які були закладені Джоном фон Нейманом (прінстонська архітектура). За цією концепцією щоб комп'ютер був ефективним і універсальним інструментом він має включати такі компоненти:

- арифметико-логічний пристрій;
- пристрій управління;
- запам'ятовуючий пристрій чи пам'ять;
- пристрої введення-виведення інформації [2].

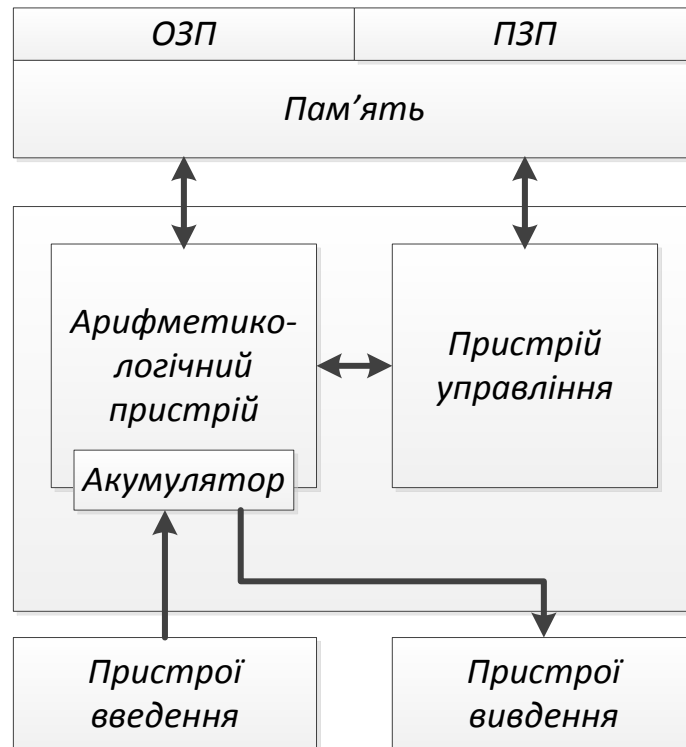


Рис. 1.1. Архітектура комп'ютера (фон Неймана)

**Арифметико-логічний пристрій** виконує арифметичні та логічні перетворення даних, що надходять до нього [2]. Наприклад, зчитування команди з комірки пам'яті А, зчитування даних з комірки пам'яті В, складення 2 чисел, віднімання, ділення, запис результату в комірку С та інші.

**Пристрій управління** автоматично керує процесом оброблення інформації, посылаючи всім іншим пристроям сигнали про виконання тих чи інших дій [2].

Сукупність арифметико-логічного пристрою та пристрою управління називають **процесором** або CPU (Central Processing Unit).

**Акумулятор** (або реєстр) також є частиною процесора і слугує для збереження проміжних результатів обчислень, щоб не звертатися кожного разу до основної пам'яті.

Крім того, процесор містить тактовий генератор, який продукує сигнали для синхронізації роботи всіх мікросхем ("гігагерци", за якими вимірюється швидкодія комп'ютера).

Тобто, коли кажуть, що процесор має частоту 2 GHz – це значить, що він генерує 2 мільярди тактових сигналів в секунду і теоретично виконує 2 мільярди елементарних операцій в секунду.

Але більшість команд є складнішими і можуть потребувати по декілька тактів, тому така характеристика швидкодії є приблизною.

**Пам'ять** зберігає програми та дані, що передані з інших пристроїв (зокрема, пристроїв уведення), і видає інформацію іншим пристроям комп'ютера, включаючи пристрої виведення.

Там же зберігаються всі проміжні дані, необхідні для виконання програми. Пам'ять складається з великої кількості комірок, кожна з яких має адресу і містить 1 байт (тобто 8 біт інформації). Це дозволяє в будь-який момент часу звернутися до будь-якої комірки і зчитати або записати туди дані. Але дані зберігаються лише поки наявне живлення, а при знеструмленні комп'ютера всі данні стираються. Саме тому ми знаємо її як оперативну пам'ять або RAM (random access memory).

Для того, щоб дані нікуди не зникали після завершення роботи комп'ютера, використовуються додаткові запам'ятовуючі пристрої з нижчою швидкістю доступу до інформації. Найбільш розповсюдженим варіантом є **жорсткий диск** – на якому зберігаються всі аудіо-, відео- та ін. файли та записана операційна система і всі інші програми. Під час запуску програми вона копіюється у оперативну пам'ять і далі виконується звідти. Всі дані, з якими працює програма, також завантажуються з жорсткого диска та назад за вимогою. Саме тому, при вимкненні комп'ютера слід зберегти всі відкриті документи – в цей час всі зміни існують лише в оперативній пам'яті і зникають з неї разом із струмом.

Жорсткий диск, як і всі інші пристрої, що підключаються до комп'ютера, відноситься до **пристроїв вводу-виводу**. Кожен з них має власний набір команд для взаємодії з комп'ютером і потребує для роботи спеціальну програму, що називається **драйвером**. Якщо в операційній системі встановлений відповідний драйвер, вона здатна "спілкуватися" з підключеним пристроєм, що дозволяє його використовувати. Найбільш розповсюджені пристрої мають стандартизовані набори команд і драйвери для них вже включені в операційну систему.

Взаємодію між усіма компонентами забезпечує так звана **системна шина**. До неї підключається все обладнання, і вона передає сигнали між окремими пристроями. Фізично вона

розташовується на материнській платі, навкруги якої і збирається сучасний комп'ютер.

Джон фон Нейман також відзначав, що комп'ютер має працювати з двійковими числами, бути електронним, а не механічним пристроєм, і виконувати операції послідовно одну за одною.

## 1.2. Подання даних

Зазвичай вхідні і вихідні дані подаються у формі, зручній для людини. Числа люди звикли зображати в десятковій системі числення. Для комп'ютера зручніше двійкова система. Це пояснюється тим, що технічно набагато простіше реалізувати пристрої (наприклад, запам'ятовуючий елемент) з двома, а не з десятьма стійкими станами (є електричний струм – немає струму, намагнічений – не намагнічений і т.п.). Можна вважати, що один з двох станів означає одиницю, інший – нуль.

Будь-які дані (числа, символи, графічні та звукові образи) в комп'ютері представляються у вигляді послідовностей з нулів і одиниць. Ці послідовності можна вважати словами в алфавіті  $\{0,1\}$ , так що оброблення даних всередині комп'ютера можна сприймати як перетворення слів з нулів і одиниць за правилами, зафіксованим в мікросхемах процесора.

Під час зчитування документів, текстів програм та інших матеріалів введені букви кодуються відповідними числами, а у разі виведення їх для читання людиною (на монітор, принтер та інше) за кожним числом будується зображення символу. Відповідність між набором символів і їх кодів називають кодуванням символів (symbolic coding).

Зазвичай код символу зберігається в одному байті. Код символу розглядається як число без знаку і, отже, може набувати значень від 0 до 255. Такі кодування називають однобайтовими; вони дозволяють використовувати до 256 різних символів. Тепер дедалі більшого поширення набуло двобайтове кодування Unicode, за якого коди символів можуть набувати значень від 1 до 65535. У цьому кодуванні є номери для майже всіх застосовуваних символів (букв та ієрогліфів різних мов, математичних, декоративних символів тощо).

**Система числення** – це сукупність прийомів та правил для зображення чисел за допомогою цифрових символів (цифр), що мають визначені кількісні значення (числовий еквівалент) [1].

Загалом, у довільній системі числення запис числа називається кодом і в скороченому вигляді може бути зображений так:

$$A = a_n a_{n-1} \dots a_2 a_1 a_0$$

Окрему позицію запису числа називають **розрядом**, а номер позиції – **номером розряду**.

Залежно від способів зображення чисел цифрами та способу визначення числового еквіваленту системи числення поділяють на дві групи: **непозиційні і позиційні**.

**Непозиційна система числення** – це така система, в якій значення символу (значення числового еквіваленту) не залежить від його позиції (розряду) в записі числа, а залежить лише від самого числа [1].

Прикладом такої системи є римська система числення, у якій цифри позначаються буквами латинського алфавіту:

$$I = 1; V = 5; X = 10; L = 50; C = 100; D = 500; M = 1000$$

Число XXX (тридцять у десятковій системі) містить у всіх розрядах один і той самий символ X, що має числовий еквівалент 10 одиниць у десятковій системі числення незалежно від його позиції у запису цього числа.

Кажуть, що цифра не змінює свою вагу залежно від позиції. Для запису проміжних значень застосовується правило: кожне менше значення символу, поставлене праворуч від більшого, додається до більшого значення, а поставлене ліворуч від більшого – віднімається від нього. Наприклад, IX – 9; XI – 11; MCMXCVIII – 1998.

**Позиційна система числення** – це така система, в якій значення символу (числовий еквівалент) залежить від його місця в записі числа [1].

Будь-яка позиційна система числення характеризується основою.

**Основа** або **базис**  $d$  натуральної позиційної системи числення – це впорядкована послідовність скінченного набору

знаків або символів, які використовуються для зображення числа у цій системі, у якій значення кожного символу залежить від його позиції (розряду) у зображенні числа.

Не існує максимальної основи системи числення, проте існує мінімальна основа системи числення, що дорівнює 2. Отже, якщо за основу можна прийняти будь-яке число (крім одиниці), то можна створити нескінченну множину позиційних систем числення.

Позиційні системи числення поділяють на дві групи: *змішані* та *однорідні*.

**Змішана позиційна система числення** – це така система, в якій кількість допустимих цифр для різних розрядів (позицій) різна [1]. Вага кожного розряду визначається як добуток ваги попереднього розряду на вагу цього розряду.

$$a_n p_n p_{n-1} \dots p_0 + a_{n-1} p_{n-1} \dots p_0 + \dots + a_0 p_0$$

Прикладом змішаної системи є система вимірювання часу.

Якщо прийняти, що  $d_0 = 1\text{с}$ ;  $d_1 = 60\text{с}$ ;  $d_2 = 60\text{хв}$ ;  $d_3 = 24\text{год}$ , то в цій системі запис часу 10 діб 11 годин 35 хв 50 с в секундах матиме вигляд:

$$10 \cdot 24 \cdot 60 \cdot 60 \cdot 1 + 11 \cdot 60 \cdot 60 \cdot 1 + 35 \cdot 60 \cdot 1 + 50 \cdot 1$$

**Однорідна позиційна система числення** – це така позиційна система числення, в якій є одна основа  $d$ , а вага  $i$ -го розряду дорівнює  $p^i$  [1].

**Вага** розряду  $p^i$  числа у позиційній системі числення – це відношення  $p_i = d^i / d^0 = d^i$ , де  $i$  – номер розряду справа наліво, а  $d^0$  – це перший розряд ліворуч від коми і його номер дорівнює 0, а значення дорівнює 1.

Ціле число у однорідній позиційній системі числення у загальному випадку записується як поліном у такому вигляді:

$$a_n p^n + a_{n-1} p^{n-1} + \dots + a_1 p^1 + a_0 p^0,$$

а дробове:

$$a_{-1} p^{-1} + a_{-2} p^{-2} + \dots + a_{-m} p^{-m}$$

Отже, кожне число у позиційній системі числення з основою  $d$  може бути записане у вигляді дискретної суми степенів основи системи з відповідними коефіцієнтами [1]:

$$A_d = \sum_{i=1}^{-m} a_i d^i = a_n d^n + a_{n-1} d^{n-1} + \dots + a_1 d^1 + a_0 d^0 + a_{-1} d^{-1} + a_{-2} d^{-2} + \dots + a_{-m} d^{-m} \quad (1)$$

де  $A_d$  – довільне число у системі числення з основою  $d$ ;

$a_i$  – коефіцієнти ряду або цифри системи числення;

$i=(n, n-1, n-2, \dots, 1, 0, -1, \dots, -m+1, -m)$  – номер розряду цілої ( $n$ ) або дробової ( $-m$ ) частини числа.

### **Десяткова система числення**

Найперші обчислення в історії людства проводилися на пальцях. Тому число 10 лежить в основі десяткової системи числення та всіх наших розрахунків.

Основу десяткової системи становлять символи 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Отже,  $a_i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ , основа системи  $d = 10$ . Кожна позиція оцінюється значенням або вагою (одиниці, десятки, сотні тощо). Так, у десятковій системі числення у числі перша цифра праворуч означає кількість одиниць, друга цифра – кількість десятків, третя цифра – означає кількість сотень і т.д.

Присвоєння значення кожній позиції, наприклад, десяткового числа 7268 можна наочно записати так:

<b>Число</b>	<b>7</b>	<b>2</b>	<b>6</b>	<b>8</b>
<b>Позиція</b>	3	2	1	0
<b>Вага розряду</b>	$10^3$	$10^2$	$10^1$	$10^0$

Відповідно, це число можна записати у вигляді:

$$7268 = 7 \cdot 10^3 + 2 \cdot 10^2 + 6 \cdot 10^1 + 8 \cdot 10^0$$

Якщо десяткове число має дробову частину, то її відокремлюють від цілої частини комою або крапкою. Розгорнута форма запису дробової частини числа буде мати, наприклад, для числа 0,243, такий вигляд:

$$0,243 = 2 \cdot 10^{-1} + 4 \cdot 10^{-2} + 3 \cdot 10^{-3}$$

### ***Двійкова система числення***

Двійкова система числення у комп'ютерах є основною, у якій здійснюються арифметичні і логічні перетворення інформації у пристроях комп'ютера. Вона має тільки дві цифри: 0 і 1, а всяке двійкове число зображується у вигляді комбінації нулів і одиниць відповідно до виразу (1).  $d = 2$  і  $a_i = 0, 1$  [1, 2]. Будь-яке двійкове число може бути зображене за допомогою формули розкладу десятковим еквівалентом, наприклад:

$$11011,012 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 16 + 8 + 2 + 1 + 0,25 = 27,25_{10}$$

Щоб не плутати числа, складені з тих самих цифр, але таких, що належать до різних систем числення рекомендують зазначати основу системи числення у вигляді підрядкового індексу. Наприклад, запис  $101,01_2$  означає, що розглядається двійкове число один нуль один кома нуль один, а запис  $101,01_{10}$  відповідає десятковому числу сто один і одна десята.

### ***Шістнадцяткова система числення***

Шістнадцяткова система числення має основу  $d = 16$  і  $a_i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$ . Для запису чисел у системі числення з основою більше ніж 10 арабських цифр виявляється недостатньо і доводиться додатково вводити символи, що однозначно представляють цифри від 10 до 15 [1, 2]. У цій системі числення застосовують великі латинські (англійські) символи для позначення цифр від 10 до 15.

Будь-яке число з шістнадцяткової системи числення також може бути зображене десятковим числом за допомогою формули розкладу, наприклад:

$$10 A, F_{16} = 1 \cdot 16^2 + 0 \cdot 16^1 + 10 \cdot 16^0 + 15 \cdot 16^{-1} = (266 \frac{15}{16})_{10}$$

Команди і дані, записані у шістнадцятковій формі, у чотири рази коротше ніж записані у двійковій формі.

### ***Переведення цілого числа з однієї системи числення до іншої***

Для того щоб ціле число перевести з однієї системи числення у іншу, число послідовно ділять на основу нової системи



числення, записану у початковій системі числення, до отримання частки, що дорівнює нулю. Число у новій системі числення записується як послідовність залишків від ділення, починаючи з останнього.

Операцію ділення виконують у початковій системі числення, тому її зручно використовувати для переведення десяткових чисел до інших систем числення.

**Приклад 1.** Перевести число  $15_{10}$  у двійкову систему числення.

$$15 \div 2 = 7 \quad (\text{залишок} = 1)$$

$$7 \div 2 = 3 \quad (\text{залишок} = 1)$$

$$3 \div 2 = 1 \quad (\text{залишок} = 1)$$

$$1 \div 2 = 0 \quad (\text{залишок} = 1)$$

**Приклад 2.** Перевести число  $15_{10}$  у вісімкову систему числення.

$$15 \div 8 = 1 \quad (\text{залишок} = 7)$$

$$1 \div 8 = 0 \quad (\text{залишок} = 1)$$

Число записується з кінця:  $15_{10} = 1111_2 = 17_8$

**Приклад 3.** Перевести числа  $11001011_2$  і  $313_8$  до десяткової системи числення.

$$1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^3 + 1 \cdot 2^1 + 1 \cdot 2^0 = 203_{10}$$

$$3 \cdot 8^2 + 1 \cdot 8^1 + 3 \cdot 8^0 = 203_{10}$$

Щоб виконати переведення правильного дробу, його множать на основу нової системи числення, записаної в початковій системі числення. Далі дробові частини від множення послідовно множать на основу нової системи числення. Операцію множення виконують в початковій системі числення. Правильний дріб у новій системі числення записується як послідовність цілих частин від множення, починаючи з першого [1, 2].

Переведення закінчується, коли проміжний добуток дорівнює 0 у всіх розрядах або досягнута необхідна точність, тобто отримана необхідна кількість розрядів результату після коми.

**Приклад 4.** Десятковий дріб  $0,3126_{10}$  перевести у двійкову систему числення з точністю до  $2^{-4}$ .

$$\begin{array}{r} \times 0,3126 \\ \hline 2 \\ \hline 0,6252 \end{array}$$

$$\begin{array}{r} \times 0,6252 \\ \hline 2 \\ \hline 1,2504 \end{array}$$

$$\begin{array}{r} \times 0,2504 \\ \hline 2 \\ \hline 0,5008 \end{array}$$

$$\begin{array}{r} \times 0,5008 \\ \hline 2 \\ \hline 1,0016 \end{array}$$

Результат буде записаний як:  $0,3126_{10} = 0,0101_2$

У разі переведення змішаних чисел з однієї системи числення у іншу необхідно окремо перевести його цілу і дробову частини за відповідними правилами, а потім обидва результати об'єднати у одне число у новій системі числення.

### ***Особливості переведення вісімкових і шістнадцяткових чисел***

Будь-яке однорозрядне вісімкове число можна записати у вигляді трирозрядного двійкового, а будь-яке трирозрядне двійкове число можна записати у вигляді однорозрядного вісімкового.

Для переведення вісімкових чисел до двійкової системи числення необхідно кожен вісімкову цифру замінити еквівалентною їй ***двійковою тріадою*** (для шістнадцяткових чисел – ***тетрадою***).

Для переведення двійкових чисел до вісімкової системи числення необхідно двійкове число розділити на тріади праворуч і ліворуч від коми (для шістнадцяткових чисел – на тетради). Якщо останні ліворуч і праворуч тріади (тетради) будуть неповні, їх потрібно доповнити нулями. Потім кожен двійкову тріаду (тетраду) замінити однією еквівалентною їй вісімковою (шістнадцятковою) цифрою [1, 2].

### ***Внутрішнє машинне представлення цілих та дійсних чисел***

Для подання чисел використовують один чи декілька послідовно розміщених байтів. Групи байтів утворюють двійкові слова, що, у свою чергу, можуть бути як фіксованої, так і змінної довжини.

Формати даних фіксованої довжини (півслово, слово і подвійне слово) складаються відповідно з одного, двох і чотирьох послідовно розміщених байтів. Звернення до цих даних виконується за адресою крайнього лівого байта числа, що для слова має бути кратним числу 2, а для подвійного слова – числу 4.

Формат даних змінної довжини складається з групи послідовно розміщених байтів від 1 до 256. Адресація таких

даних виконується, як і у форматах фіксованої довжини, за адресою найлівішого байта [2].

Залежно від характеру інформації у сучасних комп'ютерах застосовують дві форми подання чисел: з **фіксованою точкою (комою)** і з **плаваючою точкою (комою)**. Так, у форматах даних фіксованої довжини зазвичай подаються двійкові числа, команди і деякі логічні дані, а у форматах даних змінної довжини – десяткові числа, алфавітно-цифрова і деяка логічна інформація.

У разі подання чисел з фіксованою точкою (в першій формі) положення точки фіксується у певному місці відносно розрядів числа. У перших комп'ютерах точка фіксувалася перед старшим розрядом числа, тому подані числа за абсолютною величиною були менші від одиниці. У сучасних комп'ютерах точка фіксується праворуч від наймолодшого розряду і тому можуть подаватися тільки цілі числа. При цьому використовують два варіанти подання цілих чисел: **зі знаком** і **без знаку**.

Для числа зі знаком крайній розряд ліворуч потрапляє під знак числа. У цьому розряді записується нуль для додатних чисел і одиниця – для від'ємних.

Числа без знаку займають усі розряди числа, тобто числа можуть бути тільки додатними. Нумерація розрядів числа зазвичай ведеться справа наліво.

У комп'ютерах числа з фіксованою точкою мають три основні формати – один байт (півслово), 16-розрядне слово (короткий формат) і 32-розрядне подвійне слово (довгий формат).

### ***Подання цілих додатніх чисел***

Множина цілих чисел, які представлені в пам'яті комп'ютера обмежена і залежить від розміру комірок пам'яті (машинного слова), які використовуються для їх зберігання. В k-розрядній комірці може зберігатися  $2^k$  (65536) різних значень цілих чисел.

Як вже зазначалося, розряди нумеруються справа наліво, починаючи з 0.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
----	----	----	----	----	----	---	---	---	---	---	---	---	---	---	---

В даній комірці можна зберігати  $2^{16}$  цілих чисел. Діапазон допустимих значень залежить також від того, чи будуть в комірці зберігатися числа зі знаком чи без знаку.

Для запису цілого числа без знаку досить перевести його в двійкову систему числення і при необхідності доповнити результат зліва незначущими нулями [2, 3].

Додатковий код цілого додатнього числа збігається з його прямим кодом. Для його отримання необхідно:

1. Перевести число  $N$  в двійкову систему числення.
2. Отриманий результат доповнити зліва незначущими нулями до  $k$  розрядів.
3. При необхідності перевести число в стислу шістнадцяткову форму.

**Приклад 5.** Отримати внутрішнє представлення цілого числа 1607 в 2-х байтовій комірці пам'яті. Записати відповідь в 16-ій формі.

1)  $1607_{10} = 11001000111_2$

2) Внутрішнє представлення цього числа:

0000 0110 0100 0111

3) 16-а форма: 0647.

### ***Подання цілих від'ємних чисел***

Для представлення знакових цілих чисел та спрощення арифметичних операцій використовуються три способи [2, 3, 4]:

- 1) прямий код;
- 2) зворотний код;
- 3) додатковий код.

Прямий код двійкового числа містить цифрові розряди, ліворуч від яких записується знаковий розряд. Додавання в прямому коді чисел, що мають однакові знаки, виконується досить просто. Цифрові розряди чисел додаються за правилами арифметики, і сумі присвоюється код знаку доданків. Значно складніше реалізовується в прямому коді операція алгебричного додавання, тобто додавання чисел, що мають різні знаки. У цьому разі доводиться визначати більше за модулем число, вираховувати числа і присвоїти різниці знак більшого за модулем числа.

За допомогою оберненого і додаткового кодів операція віднімання (чи алгебричного додавання) зводиться до арифметичного додавання, спрощується визначення знаку результату операції, а також полегшується вироблення ознак

переповнення результату (коли в результаті арифметичних операцій число стає більшим від максимально допустимого для цієї форми значення).

Обернений код від'ємного числа одержується за таким правилом: у знаковий розряд числа записується одиниця, у цифрових розрядах нулі замінюються одиницями, а одиниці – нулями.

Додатковий код від'ємного числа отримують з оберненого коду додаванням одиниці до молодшого розряду.

Під час виконання операції алгебричного додавання з використанням оберненого чи додаткового коду додатні числа подаються прямим кодом, а від'ємні – оберненим або додатковим кодом. Потім виконується арифметичне підсумовування цих кодів, включаючи знакові розряди, що при цьому розглядаються як старші. У разі використання оберненого коду виникла одиниця перенесення зі знакового розряду циклічно додається до молодшого розряду суми кодів, а у разі використання додаткового коду ця одиниця вилучається.

Алгоритм отримання внутрішнього представлення цілого від'ємного числа  $N$ , що зберігається у  $k$ -розрядному машинному слові:

1. Отримати внутрішнє представлення додатнього числа  $N$ .
2. Отримати зворотний код цього числа заміною 0 на 1 і 1 на 0, тобто значення всіх біт інвертувати.
3. До отриманого числа додати 1 (отримати доповняльний код).
4. При необхідності записати стисле внутрішнє машинне подання.

**Приклад 6.** Отримати внутрішнє представлення цілого числа  $-1607$  в 2-х байтовій комірці пам'яті. Записати відповідь в 16-ій формі.

1) Внутрішнє представлення цього додатнього числа:

0000 0110 0100 0111

2) Зворотний код – 1111 1001 1011 1000

3) Доповняльний код – 1111 1001 1011 1001

4) Стислий 16-ий код – F9B9

У разі подання величини зі знаком найлівіший (старший) розряд вказує на позитивне число, якщо містить нуль, і на

від'ємне, якщо – одиницю. Це необхідно враховувати при зворотному перекладі.

**Приклад 7.** Дано стисле 16-е внутрішнє представлення числа – CF18. Визначити, що це за число.

1) CF18 = 1100 1111 0001 1000

2) Число від'ємне, оскільки старший розряд дорівнює 1, тому отримуємо зворотній код – 1100 1111 0001 0111 (відняти 1)

3) Прямий код – 0011 0000 1110 1000

4)  $30E8_{16} = 12520_{10}$

### **Представлення дійсних чисел**

Для подання дійсних чисел у пам'яті комп'ютера був розроблений стандарт **IEEE 754** (Institute of Electrical and Electronics Engineers – Інститут інженерів з електротехніки й електроніки) (табл. 1.3). Він використовується багатьма мікропроцесорами і програмними засобами [2, 3].

Таблиця 1.3. Подання дійсних чисел у пам'яті комп'ютера за стандартом IEEE 754

<b>Формат точності</b>	<b>Розмір</b>	<b>Мантиса</b>	<b>Експонента</b>	<b>Діапазон значень</b>	<b>Точність</b>
Половинна	2 Б	11	5	$10^{-4}..10^4$	3
Одинарна	4 Б	24	8	$10^{-38}..10^{38}$	7
Подвійна	8 Б	53	11	$10^{-307}..10^{307}$	16
Чотирикратна	16 Б	113	15	$10^{-4931}..10^{4931}$	34

Дійсні числа в пам'яті комп'ютера представлені в форматі з плаваючою десятковою комою (**експоненційній формі**) [2, 3, 4].

Будь-яке число А може бути представлене в експоненційній формі:

$$A = m \cdot q^n,$$

де m – мантиса числа, q – основа системи числення, N – порядок числа.

Наприклад:  $555,55 = 0,55555 \cdot 10^3$

Так як варіантів представлення одного і того ж числа в експоненційній формі безліч, то для внутрішнього машинного представлення домовилися представляти числа у **нормалізованій** або **денормалізованій формі**.

У **денормалізованій** формі мантиса повинна відповідати такій умові: вона повинна бути правильним дробом і мати після коми цифру, відмінну від нуля, тобто  $1 / n \leq |m| < 1$

Наприклад,  $555,55$  – звичайна форма,  $0,55555 \cdot 10^3$  – денормалізована форма. Це стосується і від'ємних чисел, тому що мантиса в умові взята по модулю.

**Нормалізована** мантиса містить свій старший біт зліва від точки. У загальному випадку мантиса повинна задовольняти умові  $1 \leq |m| < 10$ . Так як числа представлені в двійковому вигляді, то цей біт завжди дорівнює 1. Іншими словами нормалізована мантиса належить інтервалу  $1 \leq |m| < 2$ . У пам'яті машини цей біт не зберігати, тобто є "прихованим".

Для додатніх і від'ємних чисел мантиса в пам'яті представлена в **прямому коді**.

**Приклад 8.** Представити число  $155,625$  у двійковій системі числення в експоненційному нормалізованому вигляді.

$155,625_{10} = 10011011,101_2$  – число у двійковій системі.

Отримане число нормалізованого виду в десятковій і двійковій системах:

$$1,55625 \cdot 10^2 = 1,0011011101 \cdot 2^{11}$$

Мантиса  $m = 1,0011011101$

Експонента  $exp_2 = +111$

### **Обчислення машинного порядку**

Для зберігання дійсних чисел в пам'яті комп'ютера виділяються наступні розряди:

- знак числа (старший біт),
- машинний порядок числа
- мантиса.

Щоб спростити операції з порядками, їх зводять до дій над цілими додатними числами використанням зміщеного порядку, що завжди додатний. Отже, щоб не зберігати знак порядку використовується так званий **машинний порядок**. Машинний

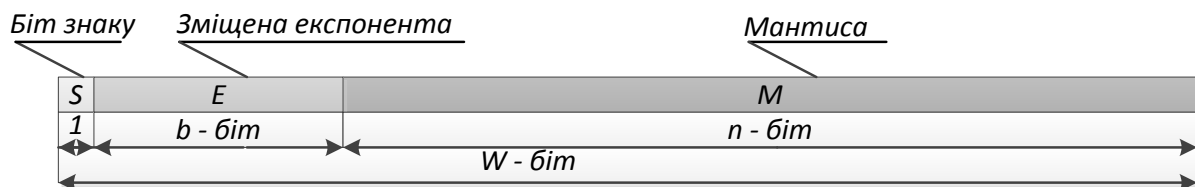
порядок зміщений відносно математичного порядку і має тільки додатні значення. Зсув вибирається так, щоб мінімальному математичному значенню порядку відповідав нуль.

Зв'язок між машинним порядком ( $M_p$ ) і математичним ( $p$ ) в даному випадку виражається формулою:  $M_p = (2^{n-1} - 1) + p$ , де  $n$  – це кількість біт, що відводяться на порядок.

**Приклад 9.** Записати машинний порядок для числа 34,48, якщо відомо, що під нього відводиться 11 біт.

- 1) Двійкове подання числа –  $100010,01111_2$
- 2) Нормалізована форма числа –  $+1,0001001111 \cdot 2^{101}$
- 3)  $M_p = 2^{10} - 1 + 5 = 1028_{10} = 10000000100_2$

Формальне представлення дійсних чисел в стандарті IEEE 754



**Приклад 10.** Записати внутрішнє машинне подання числа 155,625 для 32-бітного формату (на порядок відводиться 8 біт).

- 1)  $155,625_{10} = 10011011,101_2$  – число двійковій системі
- 2)  $1,55625 \cdot 10^2 = +1,0011011101 \cdot 2^{111}$  – нормалізований вид числа
- 3)  $2^7 - 1 + 7 = 128 - 1 + 7 = 134_{10} = 10000110_2$  – машинний порядок
- 4) 01000011 00011011 10100000 00000000 – результат

Для 64-бітного представлення чисел (double) на порядок відводиться 11 біт, для 16-бітного (real) – 5 біт, 32-бітного (single) – 8 біт, для 80-бітного (extended) – 15 біт.

### 1.3. Поняття алгоритму. Властивості алгоритмів

В 1936 англійський математик Алан Тьюрінг запропонував математичну модель обчислювальної машини, відому як машина Тьюрінга. Крім того, довів, що ця машина здатна виконати будь-які обчислення, і ввів поняття *алгоритму* – як послідовності дій, необхідних машині для досягнення результату. Сама ж



комп'ютерна програма являє собою записаний набір команд, які задають алгоритм у формі, зрозумілій для машини.

Для того, щоб навчити комп'ютер щось робити, потрібно попередньо скласти алгоритм.

Визначень поняття алгоритму дуже багато. Найбільш загальним, простим та неформальним є: **алгоритм** – це набір інструкцій, що описує, як деяке завдання може бути виконане.

**Алгоритм (algorithm)** –

1) скінченна послідовність точно визначених дій або операцій, спрямованих на досягнення поставленої мети.

Іншими словами, алгоритм – система формальних правил, що визначає зміст і порядок дій над вхідними даними і проміжними результатами, необхідними для отримання кінцевого результату при розв'язуванні задачі.

2) це будь-яка коректно визначена обчислювальна процедура, на вхід (input) якої подається деяка величина або набір величин, і результатом виконання якої є вихідна (output) величина або набір значень.

При розв'язуванні будь-якої задачі та побудові алгоритму її розв'язку звичайно беруть до уваги наявність деяких вхідних даних і мають уявлення про результат, що необхідно отримати.

### **Властивості алгоритмів**

Можна навести загальні риси алгоритму:

1. **Дискретність інформації.** Кожний алгоритм має справу з даними: вхідними, проміжними, вихідними. Ці дані представляються у вигляді скінченних слів деякого алфавіту.

2. **Дискретність роботи алгоритму.** Алгоритм виконується по кроках та при цьому на кожному кроці виконується тільки одна операція.

3. **Детермінованість алгоритму.** Система величин, які отримуються в кожний (не початковий) момент часу, однозначно визначається системою величини, які були отримані в попередні моменти часу.

4. **Елементарність кроків алгоритму.** Закон отримання наступної системи величин з попередньої повинен бути простим та локальним.

5. **Виконуваність операцій.** В алгоритмі не має бути не виконуваних операцій. Наприклад, неможна в програмі призначити значення змінній "нескінченність", така операція була би не виконуваною. Кожна операція опрацьовує певну ділянку у слові, яке обробляється.

6. **Скінченність алгоритму.** Опис алгоритму повинен бути скінченним.

7. **Спрямованість алгоритму.** Якщо спосіб отримання наступної величини з деякої заданої величини не дає результату, то має бути вказано, що треба вважати результатом алгоритму.

8. **Масовість алгоритму.** Початкова система величин може обиратись з деякої потенційно нескінченної множини. Тобто, можливість виконання алгоритмів для рішення цілого класу конкретних задач, що відповідають загальній постановці задачі.

### ***Способи написання алгоритмів***

Існує чотири способи написання алгоритмів:

- вербальний (словесний);
- алгебраїчний (за допомогою літерно-цифрових позначень виконуваних дій);
- графічний;
- з допомогою алгоритмічних мов програмування.

Словесна форма запису алгоритмів використовується в різних інструкціях, призначених для виконання їх людиною.

Алгебраїчна форма найчастіше використовується у теоретичних дослідженнях фундаментальних властивостей алгоритмів.

Графічна форма відповідно до державних стандартів (ГОСТ) на оформлення документації прийнята як основна для опису алгоритмів.

Алгоритм записаний за допомогою алгоритмічної мови програмування називається програмою. Алгоритм у такій формі може бути введений у ЕОМ і після відповідного оброблення виконаний з метою отримання шуканого результату.

## Графічний спосіб подання алгоритмів

Блок-схеми – найбільш зручний спосіб візуального представлення алгоритмів. Для того, щоб не заплутатися в численних подробицях, є сенс складати блок-схему алгоритму в декілька ітерацій: почати з найбільш загального і поступово його уточнювати.

Блок-схема – наочне графічне зображення алгоритму, коли окремі його дії (етапи) зображуються за допомогою різних геометричних фігур (блоків), а зв'язки між етапами указуються за допомогою стрілок, що сполучають ці фігури.

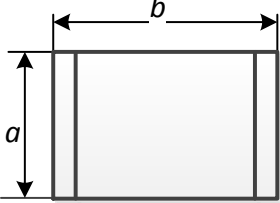
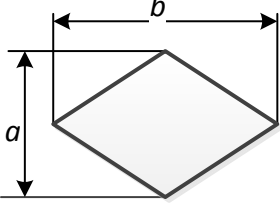
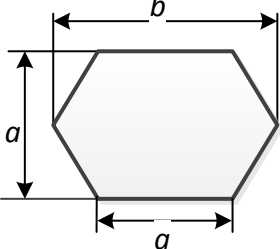
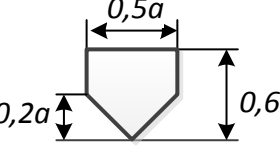
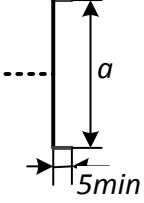
Блок-схеми відображають кроки, які повинні виконуватися комп'ютером, і послідовність їх виконання.

Умовні графічні позначення у блок-схемах алгоритмів наведені у табл. 1.4.

Таблиця 1.4. Умовні графічні позначення у блок-схемах алгоритмів

№	Назва	Графічний блок	Призначення
1	Блок початок-кінець		Вхід-вихід з програми, початок-кінець функції
2	Блок вводу-виводу даних		Уведення даних з клавіатури або виведення на екран результату
3	Обчислювальний блок		Обчислення та послідовність обчислень

Продовження таблиці 1.4.

№	Назва	Графічний блок	Призначення
4	Визначений процес		Виконання підпрограми (функції)
5	Логічний блок (блок умови)		Перевірка умови
6	Цикл		Початок циклу
7	Перехід		З'єднання між двома сторінками
8	Коментар		Пояснення

Розміри символів розраховуються відповідно до наступних правил:

- менший геометричний розмір символу слід обирати з ряду 10, 15, 20, ... мм (тобто  $a = \{10, 15, 20, \dots\}$  мм);
- співвідношення більшого та меншого розмірів має становити 1.5 (тобто  $b = 1.5 a$ ).

Початок і кінець алгоритму зображуються за допомогою овалів (табл. 1.4, підпункт 1). У середині овалу записується "початок" або "кінець".

Уведення початкових даних і виведення результатів зображуються паралелограмом (табл. 1.4, підпункт 2). Усередині нього пишеться слово "уведення" або "виведення" і перераховуються змінні, що підлягають введенню або виведенню.

Виконання операцій зображується за допомогою прямокутників (табл. 1.4, підпункт 3) в яких записано вираз/операцію. Для кожної окремої операції використовується окремий блок.

Раніше створені і окремо описані функції та підпрограми зображуються у вигляді прямокутника з бічними лініями (табл. 1.4, підпункт 4). Усередині такого "подвійного" прямокутника указуються ім'я функції (підпрограми), параметри, при яких вона повинна бути виконана.

Блок вибору, що визначає шлях, по якому підуть ці дії (наприклад, обчислення) далі, залежно від результату аналізу даних, зображується у вигляді ромбу (табл. 1.4, підпункт 5). Сама умова записується усередині ромба. Якщо умова, що перевіряється, виконується, тобто має значення "істина", то наступним виконується етап по стрілці "так". Якщо умова не виконується ("хибність"), то здійснюється перехід по стрілці "ні". Стрілки повинні бути підписані.

Шестикутник (табл. 1.4, підпункт 6) використовують для зміни параметра змінної, яка керує виконанням циклічного алгоритму, також зазначаються умови завершення циклу.

Стрілками зображуються можливі шляхи алгоритму, а малими п'ятикутниками (табл. 1.4, підпункт 7) – розриви цих шляхів потоку з переходом на наступну сторінку.

Коментарі використовуються в тих випадках, коли пояснення не поміщається усередині блока (табл. 1.4, підпункт 8).

### ***Правила графічного запису алгоритмів***

Існують такі *правила графічного запису алгоритмів*:

- блоки алгоритмів з'єднуються лініями потоків інформації;
- лінії потоків не повинні перетинатися;
- будь-який алгоритм може мати лише один блок початку і один блок кінця.

Будь-який, навіть найскладніший, алгоритм може бути поданий у вигляді комбінацій кількох елементарних фрагментів, які називають базовими структурами. До них належать:

- послідовне проходження (рис. 1.2);
- розгалуження "якщо-то" (рис. 1.3);
- розгалуження "якщо-то-інакше" (рис. 1.4);
- обирання варіанта за ключем (рис. 1.5);
- цикл з параметром (рис. 1.6);
- цикл з передумовою (рис. 1.7);
- цикл з післяумовою (рис. 1.8).
- використання коментарю (рис. 1.9).

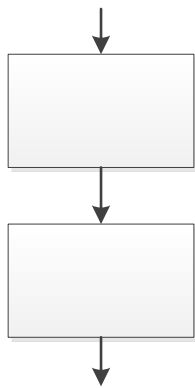


Рис. 1.2. Послідовне проходження

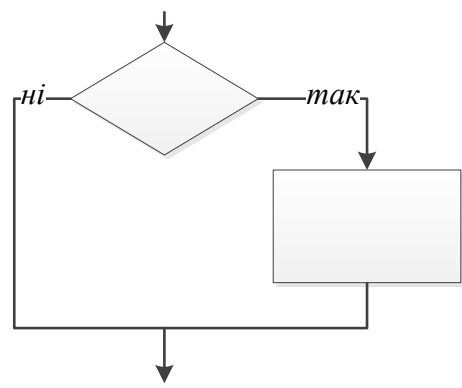


Рис. 1.3. Розгалуження "якщо-то"

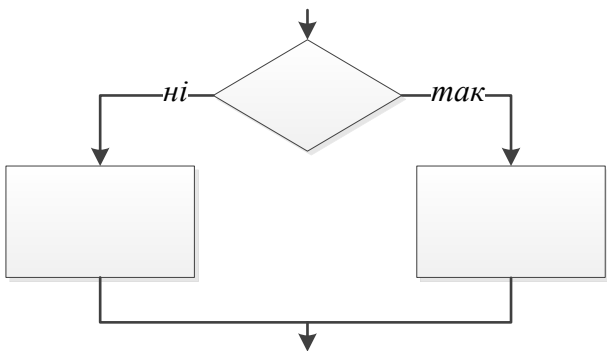


Рис. 1.4. Розгалуження "якщо-то-інакше"

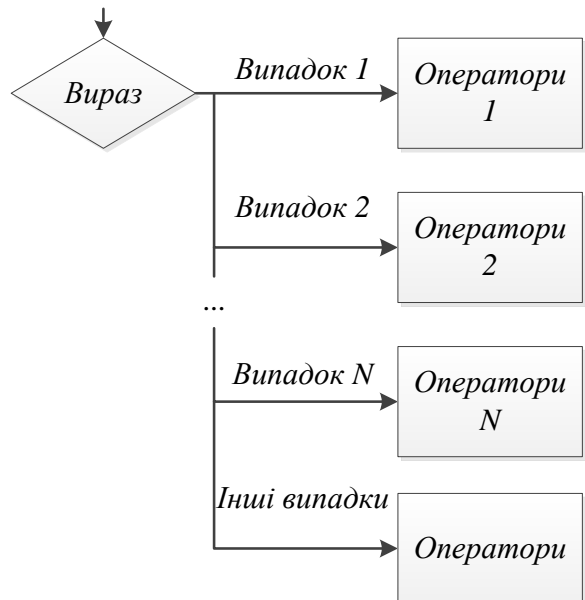


Рис. 1.5. Обирання варіанта за ключем

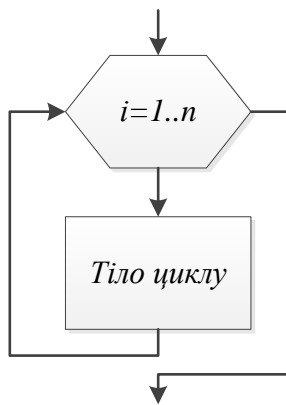


Рис. 1.6. Цикл з параметром

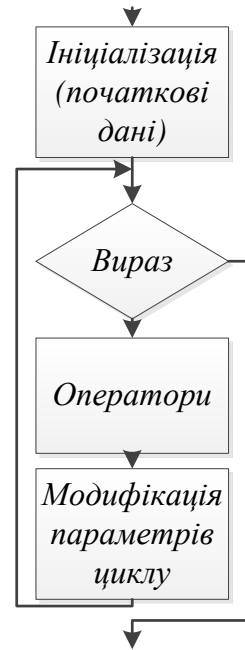


Рис. 1.7. Цикл з передумовою

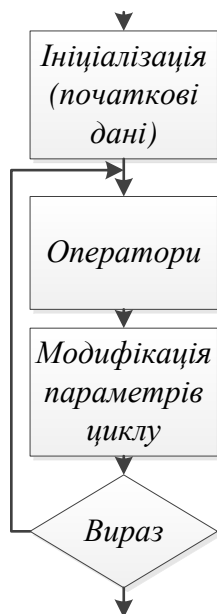


Рис. 1.8. Цикл з післяумовою

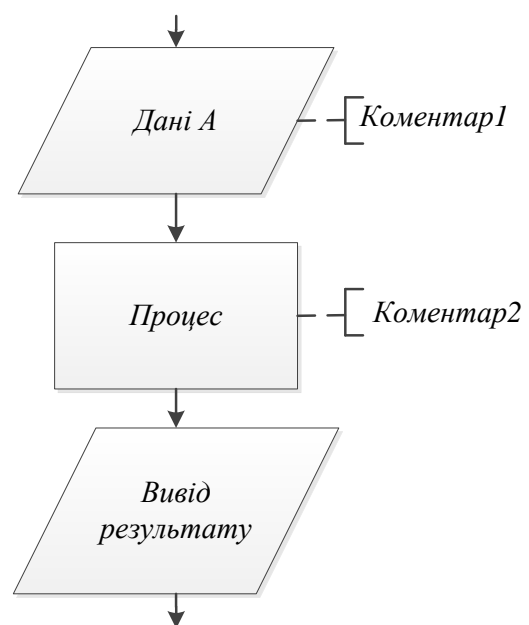


Рис. 1.9. Використання коментарю

## 1.4. Машинна програма

Отже, сам по собі комп'ютер не вміє нічого робити. Все на що він здатний це виконувати деякий набір елементарних команд, наприклад зчитати з комірки, додати значення, записати в іншу комірку. Всі можливості комп'ютера забезпечуються програмами.

**Програма** – це послідовність машинних інструкцій, що описує алгоритм.

Величезна кількість комп'ютерних програм складають **програмне забезпечення** процесу оброблення інформації. Програмне забезпечення можна розділити на три основні частини: **системне, інструментальне і прикладне** [2].

**Системне програмне забезпечення** призначене для:

- управління роботою комп'ютера;
- розподілу його ресурсів;
- підтримки діалогу з користувачем – операційні системи;
- автоматизації процесу розробки та відлагодження програм;
- перекладу мов високого рівня програмування на коди комп'ютера;
- архівація файлів тощо – утиліти;
- забезпечення роботи периферійних пристроїв – драйвери.

**Інструментальне забезпечення** слугує для розробки різних пакетів програм, що застосовуються в різних областях знання. В групу інструментальних програм входять: транслятори з різних алгоритмічних мов, які переводять текст програми на машинну мову; налагоджувачі з допомогою яких знаходять і виправляють помилки, які були допущені при написанні програми; інтегровані середовища розробки, які об'єднують вказані вище компоненти в єдину, зручну для розробки систему [2].

**Прикладне програмне забезпечення** поділяють на **прикладне програмне забезпечення загального призначення**, до якого відносяться ті програми, які широко використовуються різними категоріями користувачів (текстові редактори, електронні таблиці, системи управління базами даних, графічні редактори) та **прикладне програмне забезпечення спеціального призначення**, до якого відносяться програми, які мають специфічне призначення (засоби розробки програм, статистичні обчислення, мережеві застосунки тощо) [2].

У категорію **засобів розробки програм** відносять всі програми, що використовуються для розробки нових програм. Спектр засобів підготовки програм містить **редактори** вихідних текстів (зазвичай забезпечують підсвітку (виділення деяких елементів тексту, що мають значення для користувача: дужки, службові слова та ін.) і деяку поверхневу перевірку синтаксису



конструкції що вводяться), *транслятори* (дозволяють запускати програми), *налагоджувачі* (призначені для пошуку помилок в програмах) і в деяких випадках ще *тести* (профайлери), що дозволяють, наприклад, визначити найбільш повільний або вимогливий до ресурсів блок програми.

### 1.5. Мови програмування

Для подання алгоритму у вигляді, зрозумілому комп'ютеру, служать мови програмування. Спочатку завжди розробляється алгоритм дій, а потім він записується однією з таких мов. У підсумку виходить текст програми – повний, закінчений і детальний опис алгоритму мовою програмування.

Потім цей текст програми спеціальними службовими додатками, які називаються *трансляторами*, або переводиться в машинний код, або виконується.

*Мови програмування* – штучні мови. Від природних вони відрізняються обмеженою кількістю "слів", значення яких зрозуміло транслятору, і дуже строгими правилами запису команд-операторів. Сукупність подібних вимог формує *синтаксис* мови програмування, а сенс кожної команди та інших конструкцій мови – його *семантику* [2].

Отже, програма, з якою працює процесор, являє собою послідовність чисел, яку називають *машинним кодом*. Такий запис містить лише номери команд процесора, необхідні дані та адреси комірок пам'яті. Наприклад (для зручності двійкові дані найчастіше записуються у шістнадцятковій формі, де 2 символи відповідають 1 байту даних – шістнадцяткова система числення є досить популярною у програмуванні):

BB 11 01 B9 0D 00 B4 0E 8A 07 43 CD 10 E2 F9 CD 20 48 65 6C 6C

Для написання таких програм застосовуються мови асемблера, які дозволяють записувати команди замість числової форми у текстовій (MOV, ADD, IN, OUT) та містять деякі найпростіші засоби для полегшення написання програми. Тим не менше, навіть в такому "прикрашеному" вигляді програма залишається надзвичайно близькою до машинного коду і тому мови асемблера відносять до низькорівневих мов програмування (тобто таких, що близькі до рівня машинного коду).

Незважаючи на незручність написання, такий код виконується з найвищою швидкістю і може бути максимально оптимізований, так як програміст має доступ буквально до кожного біту у ньому. Тому найчастіше низькорівневі мови застосовуються для програмування мікросхем та окремих дій у прикладних програмах, де швидкодія є критичною.

**Асемблер** – це програма, яка перетворює код, написаний мовою асемблера, остаточно у машинний код [2]. Але часто і саму мову називають скорочено "асемблером".

Проте, більшість програм пишеться на високорівневих алгоритмічних мовах програмування. Такі мови зазвичай мають складний синтаксис, використовують слова-оператори близькі до людської мови і – що найголовніше – реалізують алгоритмічні структури для простого і зрозумілого запису програми.

З іншого боку, для виконання комп'ютером така програма має бути перетворена на машинний код або хоча б переписана низькорівневою мовою (а далі вже асемблер забезпечить її розуміння машиною). Причому команди високорівневої мови програмування можуть бути досить складними і відповідати кільком або навіть кільком десяткам машинних команд. Цей процес перетворення називається **трансляцією**, а програми, які його виконують – **трансляторами**.

Існує два типи трансляторів, що перетворюють вихідний код програм в машинні команди: **інтерпретатори** та **компілятори**.

**Інтерпретатор** зчитує вихідний код програми по одній інструкції і, в найпростішому випадку, одразу намагається їх "перекладати" та виконувати. Це дозволяє програмісту швидше перевіряти виконання програми та знаходити помилки в коді. Крім того, така програма може бути легко перенесена на іншу машину і, якщо там є потрібний інтерпретатор, виконана ним – незалежно від операційної системи та процесора. А різницю між особливостями різних комп'ютерів покриває сам інтерпретатор, який, звичайно, буде трохи відрізнятися.

Логічно, що в такій схемі виконання програми буде займати трохи більше часу – так як при цьому кожного разу відбувається аналіз коду та його перетворення.

Тому для підвищення швидкодії більшість сучасних інтерпретаторів насправді працює за змішаною схемою, спочатку

трансляючи вихідний код програми у деяку проміжну форму – так званий *байт-код*. Він є кодом нижчого рівня і ближчий до асемблеру, але машинно-незалежний – тому виконується не безпосередньо комп'ютером, а деякою віртуальною машиною, яка входить до складу інтерпретатора. Це дозволяє, за відсутності змін в оригінальній програмі, не перечитувати її повністю, а використовувати байт-код як "напівфабрикат" для роботи. Виконання байт-коду все одно повільніше ніж машинного коду, але такий підхід є компромісом, що намагається поєднати переваги інтерпретації та компіляції.

На відміну від інтерпретаторів, **компілятор** повністю перетворює вихідний код програми в машинний код, який операційна система може виконати самостійно. Це дозволяє виконувати скомпільовані програми навіть на тих комп'ютерах, на яких немає компілятора.

Проте, скомпільована програма прив'язується до операційної системи і набору команд процесора, тому не завжди може бути перенесена і виконана на іншому комп'ютері.

Крім того, такі програми виконуються швидше за рахунок того, що комп'ютеру не доводиться кожен раз перед запуском програми виконувати її розбір і перетворення в зрозумілий для себе вигляд.

Однак, при сучасних потужностях комп'ютерів і обсягах пам'яті різниця в швидкості виконання програм інтерпретаторами і компіляторами вже майже непомітна, але процес розробки і налагодження програм на інтерпретованих мовах набагато простіший.

**Програмування** – досить складний процес, і цілком природно, коли програміст припускається помилки. Так повелося, що програмні помилки називають "багами" (від англ. bug – жучок). Процес виявлення і усунення помилок в англійській літературі прийнято позначати терміном *debugging*.

Процес пошуку помилок в програмі називається **тестуванням** (*testing*), процес усунення помилок – **налагодженням** (*debugging*).

Уміння налагоджувати програми є дуже важливим навиком для програміста. Процес налагодження вимагає великих

інтелектуальних зусиль і концентрації уваги, проте це одне з найцікавіших занять.

Налагодження дуже нагадує роботу дослідника. Вивчаючи результати свого попереднього експерименту, робляться деякі висновки, потім відповідно до них змінюється програма, запускається і знову аналізується отриманий результат.

Якщо отриманий результат не співпадає з очікуваним, то необхідно знову розбиратися в причинах, які призвели до цієї невідповідності. Якщо ж гіпотеза виявиться правильною, то можна передбачити результат модифікацій програми і на крок наблизитися до завершення роботи над нею або, можливо, ще більше повірити в помилку.

Тому для перевірки працездатності програми мало перевірити її один раз – потрібно придумати всі можливі набори вхідних даних, які можуть якось вплинути на стійкість системи. Такі набори вхідних даних називають *граничними значеннями*.

Іноді процес написання і налагодження програм поділяють не тільки в часі, але і між учасниками команди розробників. Останнім часом все більшої популярності набувають так звані *гнучкі методології розробки*. У них кодування не відділяється від налагодження: програмісти, які пишуть код, також відповідають і за підготовку тестів і виявлення якомога більшої кількості помилок вже в процесі кодування.

Отже, програмування – це процес поступового доопрацювання і налагодження доти, поки програма не робитиме те, що необхідно. Починати варто з простої програми, яка робить щось просте, а потім можна приступати до нарощування її функціональності, роблячи невеликі модифікації і налагоджуючи додані частини коду.

### ***Типи помилок***

Існує три типи помилок, які можуть виникнути в програмах: ***синтаксичні помилки, помилки виконання і семантичні помилки.***

Будь-який інтерпретатор зможе виконати програму тільки в тому випадку, якщо програма синтаксично правильна. Відповідно компілятор теж не зможе перетворити програму в машинні інструкції, якщо програма містить синтаксичні помилки. Коли

транслятор знаходить помилку (тобто доходить до інструкції, яку не може зрозуміти), він перериває свою роботу і виводить повідомлення про помилку.

Другий тип помилок зазвичай виникає під час виконання програми (їх прийнято називати *винятковими ситуаціями* або, коротко – *винятками*, англ. exceptions). Такі помилки мають іншу причину. Якщо в програмі виникає виняток, то це означає, що по ходу виконання сталося щось непередбачене: наприклад, програмою було передано некоректне значення, або програма спробувала розділити якийсь значення на нуль, що є неприпустимим з точки зору дискретної математики. Якщо операційна система надсилає запит на негайне завершення програми, то також виникає виняток.

Третій тип помилок – *семантичні помилки*. Першою ознакою наявності у програмі семантичної помилки є те, що вона виконується успішно, тобто без виняткових ситуацій, але робить не те, що від неї очікується.

У таких випадках проблема полягає в тому, що семантика написаної програми відрізняється від того, що ви мали на увазі. Пошук таких помилок – завдання нетривіальне, тому що доводиться переглядати результати роботи програми і розбиратися, що програма робить насправді.

## 1.6. Особливості мови програмування Python

*Python* – молода сценарна мова, історія якої почалася в 1990 році, коли співробітник голандського інституту CWI, тоді ще мало кому відомий Гвідо ван Росум приймав участь в проєкті створення мови ABC. Ця мова була призначена для заміни мови BASIC в навчанні студентів основних концепцій програмування.

Паралельно з роботою над основним проєктом Гвідо ван Росум вдома на своєму Macintosh написав інтерпретатор іншої простої мови але деякі принципи мови ABC все ж були запозичені.

На честь англійського колективу комічних акторів (яких дуже любляв Гвідо) "Monty Python's Flying Circus" було названо мову та почалося її розповсюдження мережею Internet.

Мова почала швидко розвиватися, оскільки з'явилася велика кількість людей, що були зацікавлені та розумілися в розвитку мов програмування. Спочатку це була досить проста мова, невеликий інтерпретатор, незначна кількість функцій, об'єктно-орієнтоване програмування було відсутнім, але дуже швидко все це з'явилося та до сьогоднішнього дня продовжується її розвиток та виходять нові версії, де кожна наступна має декілька суттєвих відмінностей від попередньої.

Інтерпретатори Python існують під всі можливі платформи: Windows, UNIX та ін. Всі вони розповсюджуються безкоштовно.

Python є однією з десяти найпопулярніших мов програмування. Можна знайти велику кількість додатків, написаних на Python, наприклад:

- командний рядок на моніторі або у вікні терміналу;
- призначені для користувача інтерфейси, включаючи мережеві;
- веб-додатки, як клієнтські, так і серверні;
- бекенд-сервери, що підтримують великі популярні сайти;
- хмари (сервери, керовані сторонніми організаціями);
- додатки для мобільних пристроїв;
- додатки для вбудованих пристроїв;
- призначені для роботи з xml/html файлами;
- додатки призначені для роботи з http запитами;
- призначені для роботи із зображеннями, аудіо та відео файлами;
- додатки призначені для роботи з математичними та науковими розрахунками.

Хоча Python є досить швидким для більшості застосунків, проте його швидкості може виявитися не завжди недостатньою. Якщо програма проводить більшу частину часу за обчисленнями ("обмежена швидкодією процесора" (CPU-bound)), то мови C, C++ або Java впораються із завданням набагато краще, ніж Python. Але не завжди. Іноді більш якісний алгоритм (покрокове рішення) для Python перевершує за швидкістю неефективний алгоритм для C. Більш висока швидкість розробки для Python дає більше часу для експериментів над альтернативними рішеннями.

Стандартний інтерпретатор Python написаний на C і може бути поліпшений за допомогою додаткового коду. Інтерпретатори для Python стають швидшими. Мова Java була дуже повільною, коли тільки з'явилася, і для її прискорення було витрачено багато часу і грошей. Мовою програмування Python не володіє ні одна корпорація, тому він поліпшується більш плавно.

Найбільша проблема, з якою можна наразі зіткнутися, – це вибір однієї з двох існуючих версій Python. Остання версія Python 2 має номер 2.7, вона ще довго буде підтримуватися, але версія Python 2.8 ніколи не вийде. Нова розробка буде вестися лише на Python 3.

Python – інтерпретована мова програмування, що створює байт-код (файли з розширенням *.рус*, які з'являються у папці із текстами програм під час їх виконання) для більш швидкої роботи.

### Завдання на комп'ютерний практикум

1. Проаналізувати варіант завдання. За даними варіантів завдань отримати внутрішнє представлення чисел:
  - Цілого числа без знаку в однобайтовій комірці пам'яті.
  - Цілого числа без знаку в 2-х байтовій комірці. Записати відповідь у 16-ій формі.
  - Цілого числа зі знаком в 2-х байтовій комірці. Записати відповідь у 16-ій формі.
  - Представити число в двійковій системі числення в експоненційному нормалізованому вигляді
2. Скласти блок-схему алгоритму обчислення значень за даними варіантів завдань. Побудувати блок-схему у середовищі Microsoft Visio.

$Y = \sqrt{x^3 + ax^2 + bx + c};$	$x=0,35; a=x+0.52b; b=cx^2+1;$ $c=0,8$
-----------------------------------	---

### Запитання для самоконтролю

1. Що таке архітектура комп'ютера?
2. Що таке програмне забезпечення комп'ютера?
3. Що таке апаратне забезпечення комп'ютера?

4. Яке призначення головного процесора комп'ютера?
5. Для чого призначене системне програмне забезпечення комп'ютера?
6. Для чого призначене прикладне програмне забезпечення комп'ютера?
7. Що прийнято називати основою системи числення?
8. Наведіть приклади позиційних та непозиційних систем числення.
9. Наведіть правила переведення правильного дробу з однієї системи в іншу.
10. Яким чином здійснюється переведення цілого числа з десяткової системи числення в систему з основою?
11. Що називається "машинним зображенням" числа?
12. Які існують формати подання чисел у комп'ютері?
13. Як зображуються числа у нормальній формі?
14. Поясніть процес нормалізації чисел з рухомою комою.
15. Як позначають знаки у кодах чисел?
16. Як створюється обернений код?
17. Що таке алгоритм? У чому полягає суть побудови алгоритмів?
18. Наведіть властивості алгоритмів.
19. Які існують способи опису алгоритмів?
20. Що таке блок-схема?
21. Основні графічні елементи блок-схем, їх призначення.
22. Правила оформлення блок-схем.
23. Що таке програма?
24. Що таке мова програмування?
25. Які мови програмування активно використовуються сьогодні?
26. В чому різниця між компіляторами і інтерпретаторами?
27. Які типи помилок можуть виникнути в програмах?
28. Що є помилками виконання?
29. Що таке синтаксичні та семантичні помилки?
30. Мова програмування Python відноситься до інтерпретованих чи компільованих мов програмування?
31. Для яких цілей доцільно використовувати мову програмування Python?



## РОЗДІЛ 2. БАЗОВІ ПОНЯТТЯ МОВИ PYTHON

### 2.1. Базовий синтаксис

Мова Python має багато спільного з такими мовами як Perl, C та Java. Однак, є і деякі певні відмінності.

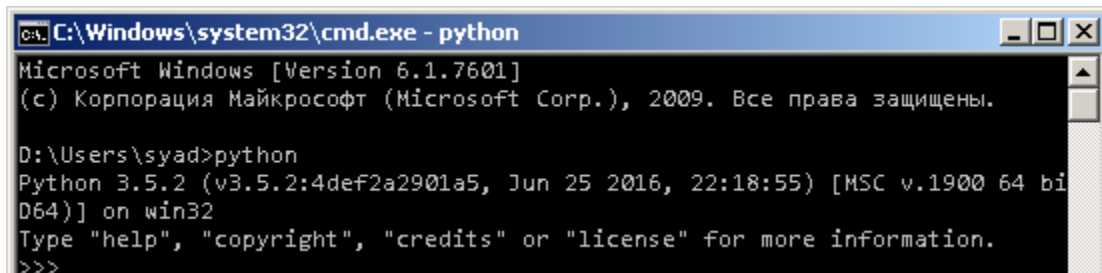
Існує два основних способи запустити програму, написану на мові Python.

Інтерактивний інтерпретатор, який поставляється разом з Python, дає можливість експериментувати з невеликими програмами. Вводячи команди рядок за рядком і миттєво отримуючи результат кожної з них.

Проте, як правило програми містять дуже велику кількість рядків коду, тож їх зберігають у вигляді текстових файлів з розширенням `.py`, а потім запускають.

#### *Інтерактивний інтерпретатор*

Працювати в інтерактивному режимі можна в консолі. Для цього слід виконати команду **python**. Запуститься інтерпретатор, де спочатку виведеться інформація про інтерпретатор. Далі, послідує запрошення до вводу (`>>>`).



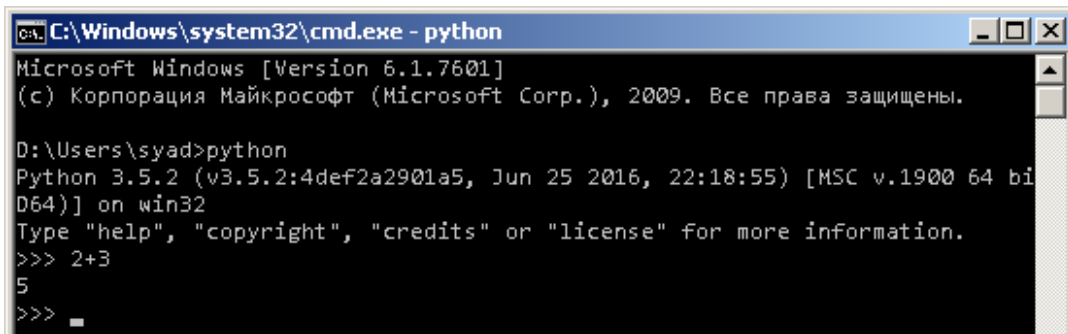
```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

D:\Users\syad>python
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bi
D64] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Рис. 2.1. Консоль. Запуск інтерпретатора

Такий режим можна використовувати для вводу інформації або для розрахунків. Вводиться команда та натискається клавіша Enter (завершення вводу команди). Після чого відразу з'являється відповідь.

Автоматичне виведення значення – це особливість інтерактивного інтерпретатора, що економить час.



```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

D:\Users\syad>python
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+3
5
>>> .
```

Рис. 2.2. Консоль. Виконання простих математичних дій

Буває, що в процесі уведення була допущена помилка або потрібно повторити раніше використовувану команду. Щоб не писати рядок спочатку, в консолі можна прокручувати список команд, використовуючи для цього стрілки на клавіатурі.

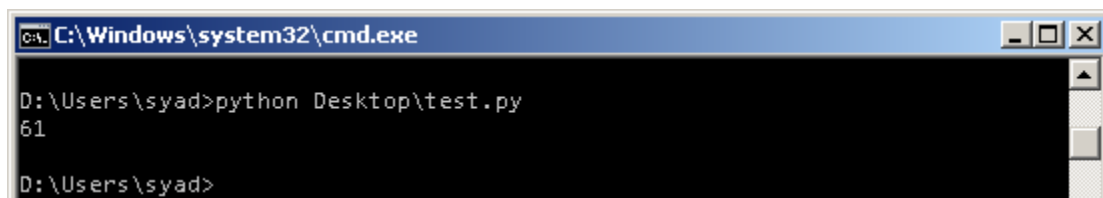
### ***Вихідний програмний код у вигляді файлу***

Незважаючи на зручності інтерактивного режиму роботи при написанні програм на Python, звичайно потрібно зберігати вихідний програмний код для подальшого використання. В такому випадку зберігаються файли, які передаються потім інтерпретатору на виконання. В інтерпретованих мовах програмування, часто вихідний код називають скриптом.

Підготувати скрипти можна у вбудованому середовищі IDLE або будь-якому редакторі редакторі. Крім того, існують спеціальні програми для розробки.

Запускати підготовлені файли можна в IDLE та в консолі за допомогою команди

**python адреса\ім'я\_файлу.py**



```
C:\Windows\system32\cmd.exe
D:\Users\syad>python Desktop\test.py
61
D:\Users\syad>
```

Рис. 2.3. Результат виконання команди, збереженої у файлі, в консолі

## 2.2. Лексеми та ідентифікатори

**Ідентифікатор** Python – це ім'я, яке використовується для ідентифікації змінної, функції, класу, модуля або іншого об'єкту.

Ідентифікатор може містити тільки такі символи:

- літери в нижньому регістрі (від "a" до "z");
- літери у верхньому регістрі (від "A" до "Z") (Python є регістро-чутливою мовою програмування);
- цифри (від 0 до 9);
- нижнє підкреслення (\_);
- не можуть співпадати з зарезервованими словами (табл. 2.2);

Ідентифікатор не може починатися з цифри.

Таким чином, *AI* та *ai* – два різних ідентифікатори в Python.

Коректними є такі імена: *a*; *ai*; *a\_b\_c\_\_95*; *\_abc*; *\_1a*.

Наступні імена є некоректними: *1*; *1a*; *1\_*.

Як правило великими літерами в Python позначаються константи.

Таблиця 2.1. Зарезервовані слова Python

false	class	finally	is	return
none	continue	for	lambda	try
true	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	Pass	
break	except	in	raise	

### **Відступи**

Код написаний мовою програмування Python не потребує фігурних дужок для позначення блоків коду для визначення класів та функцій або регулювання потоку. Блоки коду відокремлюються за допомогою рядкового відступу, якого необхідно жорстко дотримуватися.

Кількість пробілів у відступах є змінною, але всі оператори в блоці повинні бути відокремлені відступими однакової розмірності.

Наприклад:

```
if True:
    print ("True")
else:
    print ("False")
```

Однак, наступний блок коду згенерує помилку:

```
if True:
    print ("Answer")
    print ("True")
else:
    print ("Answer")
    print ("False")
```

### ***Багаторядкові речення***

Python приймає одиначні ('), подвійні (") і потрійні (' " або ") лапки, щоб позначити строкові літерали. Необхідною умовою є те що рядок має починатися і закінчуватися одним типом лапок. Однак, всередині рядка можна використовувати інші види лапок.

Потрійні лапки використовуються для розбиття рядка на кілька рядків. Наприклад:

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

### ***Коментарі в Python***

По мірі збільшення розмірів програм рано чи пізно код стане складніше читати. Для підвищення зрозумілості коду, його корисно доповнювати коментарями на природній мові, і більшість мов програмування, у тому числі Python, надають таку можливість.

Коментування коду вважається правилом "хорошого тону". Коли над програмою працює один програміст, то відсутність коментарів компенсується хорошим знанням коду, але при роботі в команді, за рідкісними винятками, коментарі просто необхідні.

**Коментар** – це фрагмент тексту у програмі, який буде проігнорований інтерпретатором Python. Можна використовувати коментарі, щоб дати пояснення до коду, зробити якісь позначки для себе, або для чогось ще. Коментар позначається символом #; все, що знаходиться після # до кінця поточного рядка, є коментарем. Зазвичай коментар розташовується на окремому рядку:

```
# Підрахунок процентного співвідношення двох величин: 20  
та 80  
print (100 * 20 / 80, "%")
```

Отримаємо:

```
25.0 %
```

Або на тому самому рядку, що і код, який потрібно пояснити:

```
print (100 * 20 / 80, "%")          # Підрахунок процентного  
співвідношення двох величин: 20 та 80
```

Отримаємо:

```
25.0 %
```

Символ # має багато імен: хеш, шарп, фунт або октоторп. Коментар діє тільки до кінця рядка, на якому він розташовується.

Однак якщо символ # знаходиться всередині текстового рядка, він стає простим символом #:

```
print("Без коментарів #")
```

Отримаємо:

```
Без коментарів #
```

### **Уведення даних**

Уведення даних з клавіатури в програму (починаючи з версії Python 3.0) здійснюється за допомогою функції *input()*. Якщо ця функція виконується, то потік виконання програми зупиняється в очікуванні даних, які користувач повинен ввести за допомогою клавіатури. Після уведення даних і натискання *Enter*, функція

*input()* завершує своє виконання і повертає результат, який є рядком символів, введених користувачем.

```
>>> input ()
1234
'1234'
>>> input ()
Hello World!
'Hello World!'
```

Коли програма, що виконується пропонує користувачеві щонебудь ввести, то користувач може не зрозуміти, що від нього хочуть. Треба якось повідомити, уведення яких саме даних очікує програма. З цією метою функція *input()* може приймати необов'язковий аргумент-запрошення строкового типу; при виконанні функції повідомлення буде з'являтися на екрані і інформувати користувача про запитовані дані.

```
>>> input("Введіть значення змінної:")
Введіть значення змінної: 25
'25'
```

Дані повертаються у вигляді рядка, навіть якщо було введено число. Якщо потрібно отримати число, то результат виконання функції *input()* змінюють за допомогою функцій *int()* або *float()*.

```
>>> input('Введіть число:')
Введіть число: 10
'10'
>>> int(input('Введіть число:'))
Введіть число: 10
10
>>> float(input('Введіть число:'))
Введіть число: 10
10.0
```

Результат, що повертається функцією *input()*, зазвичай привласнюють змінній для подальшого використання в програмі.

## ***Виведення даних***

Функція `print ()` має кілька "прихованих" аргументів, які задаються за замовчуванням в момент виклику:

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

**`print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)`**

`value` – перераховуються об'єкти через кому, що необхідно вивести на екран,

`sep` – рядок-розділювач між рядками. По замовчуванню стоїть пробіл,

`end` – рядок, що розміщено після останнього об'єкту. По замовчуванню – перехід на новий рядок,

Отже, функція `print ()` додає пробіл між кожним виведеним об'єктом, а також символ нового рядка в кінці:

```
>>> print(1, 2, 3, 4, 5)
1 2 3 4 5
>>> print(1, 6, 7, 8, 9, sep=':')
1:6:7:8:9
```

## **2.3. Змінні**

Мови програмування також дозволяють визначати ***змінні***.

***Змінна*** – це не що інше, як зарезервоване місце пам'яті для зберігання значень. Це означає, що при створенні змінної виділяється деякий простір.

Виходячи з типу даних змінної, інтерпретатор виділяє пам'ять і вирішує, що можна зберегти в зарезервованій пам'яті. Тому, призначаючи різні типи даних змінним, можна зберігати цілі числа, десяткові значення або символи в цих змінних.

Змінні в Python – це просто імена, які посилаються на значення в пам'яті комп'ютера. Можна визначити їх для використання в своїй програмі. В Python символ = застосовується для присвоювання значення змінній.

Операнд ліворуч від оператора = ім'я змінної, а операнд справа від оператора = це значення, що зберігається в змінній. Наприклад:

```
var_int = 100 # Цілочисельна змінна
var_float = 1000.0 # Десятковий дріб
var_str = "Hello" # Рядок

print (var_int)
print (var_float)
print (var_str)
```

Тут *100*, *1000.0* та *"Hello"* – привласнені значення для назв змінних *var\_int*, *var\_float* та *var\_str*, відповідно. Результатом будуть наступні значення:

```
100
1000.0
Hello
```

Присвоєння не копіює значення, воно прикріплює ім'я об'єкта, який містить дані (рис. 2.4).

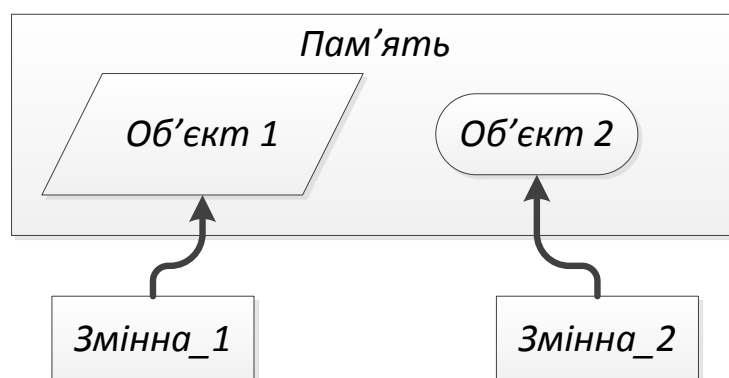


Рис. 2.4. Змінна – як посилання на об'єкт

## 2.4. Типи даних

В статичних мовах необхідно вказувати тип кожної змінної, який визначає, скільки місця змінна займе в пам'яті і що з нею



можна зробити. Комп'ютер використовує цю інформацію, щоб скомпілювати програму в дуже низькорівневу машинну мову. Оголошення типів змінних допомагає комп'ютеру знайти деякі помилки і працювати швидше, але це вимагає попереднього продумування і набору коду. Велика частина мов програмування, наприклад C, C++ і Java, вимагають оголошення типів змінних.

**Тип даних** – множина значень та множина операцій на цих значеннях. Тобто визначає можливі значення та їх сенс, операції над значеннями та способи зберігання.

**Типізація** – операція призначення типу інформаційним сутностям. Для різних мов програмування виділяють різні види типізації: статична/динамічна, сильна/слабка (табл. 2.2).

Таблиця 2.2. Групування мов програмування за типізацією

	Статична	Динамічна
Сильна	C#, Java	Python, Ruby
Слабка	C	JavaScript, PHP

При **статичній типізації** тип даних визначається на етапі компіляції. Змінна не може змінити тип, вони статичні. Ціле число – це ціле число, раз і назавжди.

При **динамічній типізації** – тип змінної визначається при призначенні їй значення. Якщо написати  $x = 5$ , динамічна мова визначить, що 5 – це ціле число, тому змінна  $x$  має тип *int*. Ці мови дозволяють досягти більшого, написавши меншу кількість рядків коду.

Динамічні мови зазвичай повільніше, ніж статичні, але їх швидкість підвищується, оскільки інтерпретатори стають більш оптимізованими. Довгий час динамічні мови використовувалися для коротких програм (сценаріїв), які часто призначалися для того, щоб підготувати дані для оброблення більш довгими програмами, написаними на статичних мовах.

**Сильна типізація** не допускає виконання операцій при несумісності типів.

**Слабка типізація** допускає виконання операцій при несумісності типів, в результаті чого можна отримати непередбачуваний результат.

Відносний лаконізм мови Python дозволяє створити програму, яка буде набагато коротшою свого аналога, написаного на статичній мові.

В Python все (цілі числа, числа з плаваючою точкою, булеві значення, рядки і різні інші структури даних, функції і програми) реалізовано як *об'єкт*. Це дозволяє Python бути стабільним, чого не вистачає деяким іншим мовам.

Python є сильно типізованою мовою – тип об'єкта не зміниться, навіть якщо можна змінити його значення.



Рис. 2.5. Типи даних

## 2.5. Прості типи даних. Числа

Числа бувають різними: *цілими, дробовими, комплексними*. Вони можуть мати величезне значення або дуже довгу дробову частину:

- **цілі числа (`int`)** – додатні і від'ємні цілі числа, а також 0 (наприклад, 4, 687, -45, 0).

- **числа з плаваючою точкою (`float`)** – дробові числа (наприклад, 1.45, -3.789654, 0.00453). Роздільником цілої і дробової частини служить точка.

- **комплексні числа (`complex`)** – зберігає пару значень типу `float`, одне з яких представляє дійсну частину комплексного числа, а інше – уявну (наприклад,  $1+2j$ ,  $-5+10j$ ,  $0.44+0.08j$ )

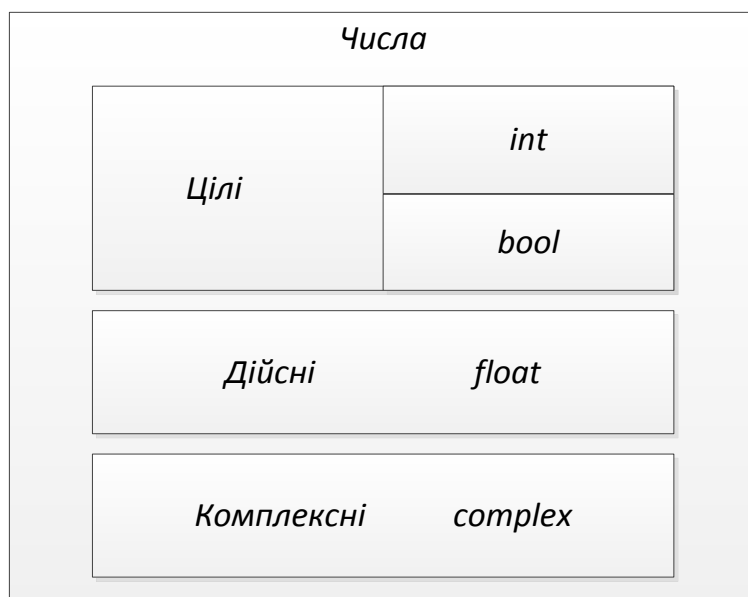


Рис. 2.6. Числові типи даних

### Операції над числами

Відомо, що **операція** – це виконання якихось дій над даними (операндами). Для виконання конкретних дій потрібні спеціальні інструменти – **оператори**, що наведені у таблиці 2.3.

Таблиця 2.3. Оператори, що застосовуються у мові програмування Python

Оператор	Опис	Приклад	Результат
+	Додавання	$5 + 8$	13
-	Віднімання	$90 - 10$	80
*	Множення	$4 * 7$	28
/	Ділення з плаваючою точкою	$7/2$	3,5
//	Цілочисельне ділення (Truncating)	$7//2$	3
%	Залишок	$7\%3$	1
**	Піднесення до степеню	$3^4$	81

Нехай змінна  $a$  має значення 20, а змінна  $b$  – значення 10:

```

a = 20
b = 10
  
```

```
c = a + b  
print ("1. Значення c = ", c)
```

```
c = a - b  
print ("2. Значення c = ", c)
```

```
c = a * b  
print ("3. Значення c = ", c)
```

```
c = a / b  
print ("4. Значення c = ", c)
```

```
c = a % b  
print ("5. Значення c = ", c)
```

```
c = a**b  
print ("6. Значення c = ", c)
```

```
c = a//b  
print ("7. Значення c = ", c)
```

Операція ділення поділяється на два підвиди:

— за допомогою оператора / виконується ділення з плаваючою точкою (десятькове ділення);

– за допомогою оператора // виконується цілочисельне ділення (ділення із залишком).

Якщо ділити ціле число на ціле число, оператор / дасть результат з плаваючою точкою.

```
1. Значення c = 30  
2. Значення c = 10  
3. Значення c = 200  
4. Значення c = 2.0  
5. Значення c = 0  
6. Значення c = 10240000000000  
7. Значення c = 2
```

Можна працювати з довільною кількістю операндів та операторів.

Вставляти пробіл між кожним числом і оператором не обов'язково. Зайві пробіли погіршують сприйняття коду:

```
print (5+4 + 7)
```

Результатом запуску даного коду буде:

```
17
```

Ділення на нуль за допомогою будь-якого оператора згенерує виняток:

```
print (a//0)
```

Результатом запуску даного коду буде:

```
Traceback (most recent call last):  
  File "G:\lab.py", line 1, in <module>  
    print (a//0)  
NameError: name 'a' is not defined
```

Якщо в одному і тому ж виразі зустрічаються декілька типів, то результат має самий сильний тип зі всіх чисел що у виразі. Самим сильним типом вважається комплексний, за ним йде дійсний, а потім цілий тип даних.

Якщо змішати чисельні значення, Python буде намагатися автоматично перетворити їх:

```
a = 20  
b = 10.8  
  
c = a + b  
print ("1. Значення c = ", c)
```

Результатом запуску даного коду буде:

```
1. Значення c = 30.8
```

Булеве значення *False* розглядається як *0* або *0.0*, коли воно змішується з цілими числами або числами з плаваючою точкою, а *True* – як *1* або *1.0*:

```
a = 20  
b = 10.8
```

```
c = a + True
print ("1. Значення c = ", c)
```

```
c = b + False
print ("2. Значення c = ", c)
```

Результатом запуску даного коду буде:

```
1. Значення c = 21
2. Значення c = 10.8
```

### ***Оператори присвоєння***

Можна поєднувати арифметичні оператори з привласненням, розміщуючи оператор перед знаком =.

Вираз `a -= 3` аналогічний виразу `a = a - 3`.

Замість знаку `-` можуть стояти інші оператори: `+`, `*`, `/`, `//`, `%`

```
a = 20
b = 10

c = a + b
print ("1. Значення c = ", c)

c += a
print ("2. Значення c = ", c)

c *= a
print ("3. Значення c = ", c)

c /= a
print ("4. Значення c = ", c)

c = 2
c %= a
print ("5. Значення c = ", c)

c **= a
print ("6. Значення c = ", c)
```

```
c //= a
print ("7. Значення c = ", c)
```

Результатом запуску даного коду буде:

1. Значення c = 30
2. Значення c = 50
3. Значення c = 1000
4. Значення c = 50.0
5. Значення c = 2
6. Значення c = 1048576
7. Значення c = 52428

### ***Пріоритет операцій***

Коли вираз містить більше одного оператора, послідовність виконання операцій залежить від порядку їх слідування у виразі, а також від їх пріоритету. Пріоритети операторів в Python збігаються з пріоритетами математичних операцій.

```
d = 2 + 3 * 4

print ("Значення d =", d)
```

Результатом запуску даного коду буде:

```
14
```

Найвищий пріоритет мають дужки, потім піднесення до степеню, множення та ділення і лише після них додавання, віднімання, далі кон'юнкція, диз'юнкція та все інше.

У випадку рівних пріоритетів розрахунок йде справа наліво. Для зміни цього порядку використовують дужки. Краще їх використовувати у всіх сумнівних ситуаціях:

```
d = (2 + 3) * 4
print ("Значення d =", d)
```

Результатом запуску даного коду буде:

```
20
```

Це спрощує візуальне сприймання виразу.

## *Перетворення типів*

Для того щоб змінити одні типи даних на інші використовуються певні стандартні функції.

Так, для зміни чогось на цілочисельний тип, слід використовувати функцію *int()*. Вона зберігає цілу частину числа і відкидає залишок.

```
a = 20
b = 10.8

q = int(a)
print ("1. Значення q = ", q)

q = int(b)
print ("2. Значення q = ", q)
```

Результатом запуску даного коду буде:

```
1. Значення q = 20
2. Значення q = 10
```

Текстовий рядок теж можна змінити на цілочисельний, якщо він буде містити цифрові символи і, можливо, знаки + і -:

```
a = "-15"
q = int(a)

print ("1. Значення q = ", q)
```

Результатом запуску даного коду буде:

```
1. Значення q = -15
```

Якщо перетворити щось несхоже на число – буде згенеровано виняток:

```
a = "xyz"
q = int(a)

print ("1. Значення q = ", q)
```



Результатом запуску даного коду буде:

```
Traceback (most recent call last):
  File "G:\lab.py", line 3, in <module>
    q = int(a)
ValueError: invalid literal for int() with base 10: 'xyz'
```

За допомогою винятків Python сповіщає про те, що сталася помилка, замість того щоб перервати виконання програми, як роблять деякі інші мови.

Останній рядок *ValueError: invalid literal for int() with base 10: 'xyz'* інформує про те, що текстовий рядок починається тими символами, які функція *int()* обробити не може.

Функція *int()* буде створювати цілі числа з чисел з плаваючою точкою або рядків, що складаються з цифр, але вона не буде обробляти рядки, що містять порожній рядок, десяткові точки або експоненти.

Для того щоб перетворити інші типи в тип *float*, слід використовувати функцію *float()*.

Перетворення значення типу *int* в тип *float* лише створить десяткову кому:

```
a = 98
b = '99'

q = float(a)
print ("1. Значення q = ", q)

q = float(b)
print ("2. Значення q = ", q)
```

Результатом запуску даного коду буде:

```
1. Значення q = 98.0
2. Значення q = 99.0
```

Можна перетворювати рядки, що містять символи, які є коректним числом з плаваючою точкою (цифри, знаки, десяткова кома чи *e*, за якою слідує експонента):

```
a = '98.6'
b = '-1.5'
```

```
c = '1.0e4'
d = True

q = float(a)
print ("1. Значення q = ", q)

q = float(b)
print ("2. Значення q = ", q)

q = float(c)
print ("3. Значення q = ", q)

q = float(d)
print ("4. Значення q = ", q)
```

Результатом запуску даного коду буде:

```
1. Значення q = 98.6
2. Значення q = -1.5
3. Значення q = 10000.0
4. Значення q = 1.0
```

## 2.6. Прості логічні вирази та логічний тип даних

В усіх мовах програмування високого рівня є можливість розгалуження програми; при цьому виконується одна з гілок програми в залежності від істинності чи хибності умови.

**Логічними виразами** називають вирази, результатом яких є істина (True) або хибність (False). У найпростішому випадку будь-яке твердження може бути істинним або хибним. Наприклад, "2 + 2 дорівнює 4" – істинний вираз, а "2 + 2 дорівнює 5" – хибний.

Розмовляючи на природній мові (наприклад, українській) порівняння позначається словами "рівно", "більше", "менше". У мовах програмування використовуються спеціальні знаки, подібні до тих, які використовуються в математичних виразах: > (більше), < (менше), >= (більше або дорівнює), <= (менше або дорівнює).

В Python використовуються наступні оператори порівняння:

- рівність (`==`);
- нерівність (`!=`);
- менше (`<`);
- менше або дорівнює (`<=`);
- більше (`>`);
- більше або дорівнює (`>=`);
- включення (`in ...`).

Ці оператори повертають булеві значення *True* або *False*.

Для перевірки на рівність використовуються два знака "дорівнює" (`==`) (один знак "дорівнює" застосовується для надання значення змінній).

```
x = 2 + 2
print('1.Результат роботи логічного виразу:', x == 4)

print('2.Результат роботи логічного виразу:', x == 5)

print('3.Результат роботи логічного виразу:', x != 5)
```

Результатом запуску даного коду буде:

```
1.Результат роботи логічного виразу: True
2.Результат роботи логічного виразу: False
3.Результат роботи логічного виразу: True
```

Результат порівняння двох значень можна записати в змінну:

```
y = x == 5
print (y)
```

Отримаємо:

```
False
```

Пріоритет операцій порівняння менший пріоритету арифметичних операцій, але більший, ніж у операції присвоювання. Це означає, що спочатку вираховується результат фрагментів, а потім вони порівнюються.

Значення *False* не обов'язково явно означає *False*. Наприклад, до *False* прирівнюються всі наступні значення:

- булева змінна *False*;

- значення *None*;
- ціле число 0;
- число з плаваючою точкою 0.0;
- порожній рядок ('');
- порожній список ([]);
- порожній кортеж (());
- порожній словник ({});
- порожня множина (set ()).

Всі інші значення прирівнюються до *True*.

## 2.7. Логічні оператори

Логічні вирази типу  $x \geq y$  є простим. Однак, на практиці не рідко використовуються більш складні. Може знадобитися отримати відповіді "Так" або "Ні" в залежності від результату виконання двох простих виразів. Для об'єднання простих виразів в більш складні використовуються **логічні оператори**: *and*, *or* і *not*.

Значення їх повністю збігаються зі значенням англійських слів, якими вони позначаються.

Щоб отримати істину (*True*) при використанні оператора *and*, необхідно, щоб результати обох простих виразів, які пов'язує цей оператор, були істинними. Якщо хоча б в одному випадку результатом буде *False* (хибність), то і весь складний вираз буде хибним.

Щоб отримати істину (*True*) при використанні оператора *or*, необхідно, щоб результати хоча б одного простого виразу, що входить до складу складного, був істинним. У разі оператора *or* складний вираз стає хибним лише тоді, коли хибні всі складові його прості вирази.

Оператор *not* унарний, тобто він працює тільки з одним операндом.

Результатом застосування логічного оператора *not* (не) відбудеться заперечення операнда, тобто якщо операнд істинний, то *not* поверне – хибність, якщо хибний, то – істину.

```
y = 6 > 8
```

```
print('y =', y)
```

```
print('1.Результат роботи логічного виразу "not y":', not y)
print('2.Результат роботи логічного виразу "not None":', not
None )
print('3.Результат роботи логічного виразу "not 2":', not 2)
```

Результатом запуску даного коду буде:

```
y = False
1.Результат роботи логічного виразу "not y": True
2.Результат роботи логічного виразу "not None": True
3.Результат роботи логічного виразу "not 2": False
```

Логічний оператор *and* поверне *True* або *False*, якщо його операндами є логічні вирази.

```
print(2>4 and 45>3)
```

Отримаємо:

```
False
```

Якщо операндами оператора *and* є об'єкти, то в результаті Python поверне об'єкт:

```
print(" and 2)
```

Отримаємо:

```
"
```

Для обчислення оператора *and* Python обчислює операнди зліва направо і повертає перший об'єкт, який має хибне значення.

```
print(0 and 3)
```

Отримаємо:

```
0
```

Якщо Python не вдається знайти хибний об'єкт-операнд, то він повертає крайній правий операнд.

```
print(5 and 4)
```

Отримаємо:

```
4
```

Логічний оператор *or* діє схожим чином, але для об'єктів-операндів Python повертає перший об'єкт, який має істинне значення. Python припинить подальші обчислення, як тільки буде знайдений перший об'єкт, який має істинне значення.

```
print(2 or 3)
```

Отримаємо:

```
2
```

Таким чином, кінцевий результат стає відомий ще до обчислення решти виразу.

```
print(None or 5)      # Повертає другий об'єкт, тому що  
перший завжди хибний  
print(None or 0)     # Повертає об'єкт, що залишився
```

Отримаємо:

```
5  
0
```

Логічні вирази можна комбінувати:

```
>>> 1+3 > 7  
False  
>>> 1+(3>7)  
1
```

В Python можна перевіряти приналежність інтервалу:

```
x=0  
print(-5<x<10)      # Еквівалентно: x > -5 and x<10
```

Отримаємо:

```
True
```

Рядки в Python теж можна порівнювати по аналогії з числами. Символи, як і все інше, представлено в комп'ютері у вигляді чисел. Є спеціальна таблиця, яка ставить у відповідність кожному символу деяке число 20. Визначити, яке число відповідає символу можна за допомогою функції *ord()*:

```
q=ord('L')
```

```
print ("1. Значення 'L' =", q)

q=ord ('Ф')
print ("2. Значення 'Ф' =", q)

q=ord ('A')
print ("3. Значення 'A' =", q)

q=ord ('a')
print ("4. Значення 'a' =", q)
```

Результатом запуску даного коду буде:

```
1. Значення 'L' = 76
2. Значення 'Ф' = 1060
3. Значення 'A' = 65
4. Значення 'a' = 97
```

Порівняння символів зводиться до порівняння чисел, які їм відповідають.

```
q='A' > 'L'
print ("Порівняння 'A' > 'L' =", q)
```

Отримаємо:

```
Порівняння 'A' > 'L' = False
```

Для порівняння рядків Python їх порівнює посимвольно:

```
q='Aa' > 'Ll'
print ("Порівняння 'Aa' > 'Ll' =", q)
```

Отримаємо:

```
Порівняння 'Aa' > 'Ll' = False
```

Оператор *in* перевіряє входження підрядка в рядок:

```
q='a' in 'abc'
print("Результат входження 'a' in 'abc' -", q)

q='A' in 'abc'          # Великої літери A немає в рядку 'abc'
print("Результат входження 'A' in 'abc' -", q)
```

```
q="" in 'abc'          # Порожній рядок міститься в будь-
якому рядку
print("Результат входження " in 'abc' "-", q)

q=" in "
print("Результат входження " in " "-", q)
```

Отримаємо:

```
Результат входження 'a' in 'abc' - True
Результат входження 'A' in 'abc' - False
Результат входження " in 'abc' - True
Результат входження " in " - True
```

## 2.8. Складні структури даних. Рядки

Завдяки підтримці стандарту Unicode Python 3 може містити символи будь-якої мови світу, а також багато інших символів. Необхідність роботи з цим стандартом була однією з причин зміни Python 2. Іноді використовуються рядки формату ASCII.

Рядки представляють собою послідовності символів.

На відміну від інших мов, в Python рядки є незмінними. Не можна змінити сам рядок, але можна скопіювати частини рядків в інший рядок, щоб отримати той же ефект.

Рядок в Python створюється заключенням символів в одинарні або подвійні лапки.

```
a = 'Hello'
b = "Hi"

print ("1. Значення a:", a)
print ("2. Значення b:", b)
```

Результатом запуску даного коду буде:

```
1. Значення a: Hello
2. Значення b: Hi
```

Два види лапок дозволяють створювати рядки, що містять лапки. У середині одинарних лапок можна розташувати подвійні і

навпаки:



```
a = "використаємо апостроф у слові подвір'я"
```

```
print ("Значення a:", a)
```

Результатом запуску даного коду буде:

```
Значення a: використаємо апостроф у слові подвір'я
```

Будь-яка програма стає більш зрозумілою, якщо її рядки відносно короткі. Рекомендована (але не обов'язкова) максимальна довжина рядка дорівнює 80 символам. Якщо є необхідність надрукувати рядок що містить більше 80 символів, використовують символ відновлення `\`, що розміщується в кінці рядка, і далі Python буде діяти так, ніби це все той же рядок.

```
alphabet = 'abcdefg' \  
           'hijklmnop' \  
           'qrstuv' \  
           'wxyz'
```

```
print ("Значення alphabet:", alphabet)
```

Результатом запуску даного коду буде:

```
Значення alphabet: abcdefghijklmnopqrstuvwxyz
```

### ***Створення керуючих символів***

Крім цього зворотний слеш (`\`) дозволяє створювати керуючі послідовності всередині рядків. Найбільш поширена послідовність `\n`, яка означає перехід на новий рядок. З її допомогою можна створити багаторядкові рядки з однорядкових:

```
alphabet = "\nabcdefg\nhijklmnop\nqrstuv\nwxyz"
```

```
print ("Значення alphabet:", alphabet)
```

Результатом запуску даного коду буде:

```
Значення alphabet:  
abcdefg  
hijklmnop  
qrstuv  
wxyz
```

Найбільш уживаними є:

`\t` – знак табуляції

`\\` – похила риса вліво

`\'` – символ одинарних лапок

`\"` – символ подвійних лапок

```
print('\tabc')
print('a\tbc')
print('ab\tc')
```

Отримаємо:

```
      abc
a      bc
ab    c
```

Якщо потрібен зворотний слеш, необхідно надрукувати два:

```
print('abc\\')
```

Отримаємо:

```
abc\
```

### ***Перетворення типів***

Подібно функціям `int()` та `float()`, можна перетворювати інші типи даних Python в рядки за допомогою функції `str()`:

```
a = 98.6
b = -1.5
c = 1.0e4
d = True

q = str(a)
print ("1. Значення q = ", q)

q = str(b)
print ("2. Значення q = ", q)

q = str(c)
print ("3. Значення q = ", q)
```

```
q = str(d)
print ("3. Значення q = ", q)
```

Результатом запуску даного коду буде:

```
1. Значення q = 98.6
2. Значення q = -1.5
3. Значення q = 10000.0
3. Значення q = True
```

### ***Об'єднання рядків***

Можна об'єднувати рядки або рядкові змінні в Python за допомогою оператора +, або розташувавши їх послідовно один за одним:

```
q='ab'+'cd'
q='ab"cd'

print("1. Результат виконання 'ab'+'cd':", q)
print("2. Результат виконання 'ab"cd':", q)
```

Отримаємо:

```
1. Результат виконання 'ab'+'cd': abcd
2. Результат виконання 'ab'+'cd': abcd
```

Python не додає пробілів при конкатенації рядків, тому потрібно явно додати пробіли:

```
a = 'Python'
b = "'s"
c = ' philosophy'
q = a + b + c

print("Результат виконання 'a + b + c':", q)
```

Отримаємо:

```
Результат виконання 'a + b + c': Python's philosophy
```

### ***Розмноження рядків***

Оператор \* можна використовувати для того, щоб розмножити рядок.

```
a='abc'  
q = 3*a  
  
print("Результат виконання '3*a:", q)
```

Отримаємо:

```
Результат виконання '3*a: abcabcabc
```

### *Звернення до символу*

Для того щоб отримати один символ рядка, задається зміщення всередині квадратних дужок після імені рядка. Зсув першого (крайнього зліва) символу дорівнює 0, наступного – 1 і т.д. Зсув останнього (крайнього праворуч) символу може бути виражено як -1, тому не потрібно рахувати, в такому випадку зміщення подальших символів дорівнюватиме -2, -3 і т.д.:

```
string= 'abcdefghijklmnopqrstuvwxy'  
  
print (string)           # Виведення всього рядку  
print (string [0])      # Виведення першого символу рядку  
print (string [1])      # Виведення другого символу рядку  
print (string [-1])     # Виведення останнього символу рядку
```

Результатом запуску даного коду буде:

```
abcdefghijklmnopqrstuvwxy  
a  
cde  
cdefghijklmnopqrstuvwxy
```

Якщо вказати зміщення, яке дорівнює довжині рядка або більше (зміщення повинно лежати в діапазоні від 0 до довжини рядка -1), буде згенеровано виняток:

```
string= 'abcdefghijklmnopqrstuvwxy'  
print (string [100])
```

Отримаємо:

```
Traceback (most recent call last):
```

```
File "G:\lab.py", line 8, in <module>
    print (string [100])
IndexError: string index out of range
```

Оскільки рядки незмінні – не можна вставити символ безпосередньо в рядок або змінити символ за заданим індексом.

```
string= 'abcdefghijklmnopqrstuvwxyz'
string [1] = 'ello'
```

Отримаємо:

```
Traceback (most recent call last):
  File "G:\lab.py", line 10, in <module>
    string[1] = 'ello'
TypeError: 'str' object does not support item assignment
```

З рядка можна вилучати підрядок (частину рядка) за допомогою функції *slice*. Визначається *slice* за допомогою квадратних дужок, зміщення початку підрядка *start* і кінця підрядка *end*, а також розміру кроку *step*.

### [start: end: step]

Деякі з цих параметрів можуть бути відсутні. У підрядок будуть включені символи, розташовані починаючи з точки, на яку вказує зміщення *start*, і закінчуючи точкою, на яку вказує зміщення *end*.

– Оператор **[:]** дозволяє взяти зріз всієї послідовності від початку до кінця.

– Оператор **[start:]** дозволяє взяти зріз послідовності з точки, на яку вказує зміщення *start*, до кінця.

– Оператор **[: end]** дозволяє взяти зріз послідовності від початку до точки, на яку вказує зміщення *end* - 1.

– Оператор **[start: end]** дозволяє взяти зріз послідовності з точки, на яку вказує зміщення *start*, до точки, на яку вказує зміщення *end* - 1.

– Оператор **[start: end: step]** дозволяє взяти зріз послідовності з точки, на яку вказує зміщення *start*, до точки, на

яку вказує зміщення *end* мінус 1, опускаючи символи, чие зміщення всередині підрядка кратне *step*.

Зміщення зліва направо визначається як 0, 1 і т.д., а справа наліво – як -1, -2 і т.д. Якщо не вказати *start*, функція буде використовувати в якості його значення 0 (початок рядку). Якщо не вказати *end*, функція буде використовувати кінець рядка. Python не включає символ, розташований під номером, який вказаний останнім.

```
string= 'abcdefghijklmnopqrstuvwxyz'

print (string [2:5])      # Виведення символів починаючи з 3-го
                          # до 5-го
print (string [2:])      # Виведення рядку починаючи з 3-го
                          # символу
print (string [:])       # Вся послідовність від початку до кінця
print (string [20:])     # Всі символи, починаючи з 20-го і до
                          # кінця
print (string [-3:])     # Останні три символи
print (string [18:-3])  # Починаючи з 18-го і закінчуючи 4 з
                          # кінця
print (string [-6:-2])  # Закінчуючи 3 з кінця
```

Результатом запуску даного коду буде:

```
cde
cdefghijklmnopqrstuvwxyz
abcdefghijklmnopqrstuvwxyz
vwxyz
xyz
stuvw
vwxyz
```

Щоб збільшити крок, необхідно вказати його після другої двокрапки.

```
string= 'abcdefghijklmnopqrstuvwxyz'
print (string [::7])      # Кожен сьомий символ з початку
                          # до кінця
```

```

print (string [4:20:3])      # Кожен 3 символ, починаючи з 4
                             та закінчуючи 19-м
print (string [19::4])      # Кожен 4 символ, починаючи з
                             19-го:
print (string [:21:5])      # Кожен п'ятий символ від
                             початку до 20-го:

```

Отримаємо:

```

ahov
ehknqt
tx
afkpu

```

Значення *end* має бути на одиницю більше, ніж реальне зміщення.

Якщо задати від'ємний крок, Python буде рухатися у зворотний бік.

```

string= 'abcdefghijklmnopqrstuvwxy'

print (string [-1::-1])      # Всі символи, починаючи з кінця
                             і закінчуючи на початку
print (string [::-1])      # Аналогічно

```

Отримаємо:

```

zyxwvutsrqponmlkjihgfedcba
zyxwvutsrqponmlkjihgfedcba

```

### ***Строкові методи та функції***

Функція *len()* підраховує символи в рядку:

```

string= 'abcdefghijklmnopqrstuvwxy'
q=len(string)

print ('Довжина рядка string =', q)

```

Отримаємо:

```

Довжина рядка string = 26

```

Довжина порожнього рядка = 0. Функцію *len()* можна застосовувати до інших послідовностей (кортежі, словники, списки).

На відміну від функції *len()* деякі функції характерні лише для рядків.

Для того щоб використовувати строкову функцію, необхідно ввести ім'я рядка, крапку, ім'я функції і аргументи, які потрібні функції:

### **рядок.функція(аргументи)**

Однією з таких вбудованих функцій є функція *split()*, що розбиває рядок на список невеликих рядків, спираючись на роздільник.

### **рядок.split('роздільник')**

**Список** – це послідовність значень, розділених комами і оточених квадратними дужками:

```
string= 'abcdefghijklmnopqrstuvwxyz'  
q = string.split(',')  
print (q)  
  
q = string.split('k')  
print (q)
```

Результатом запуску даного коду буде:

```
['a', ' b', ' c', ' d', ' e']  
['abcdefghijklmnopqrstuvwxyz']  
['abcdefghij', 'lmnopqrstuvwxyz']
```

Рядок має ім'я *letters*, а строкова функція називається *split()* і отримує один аргумент *,*. Якщо не вказати роздільник, функція *split()* буде використовувати будь-яку послідовність пробілів, а також символи нового рядка і табуляцію:

```
letters = 'a, b, c, d, e'  
q = letters.split()
```



```
print (q)
```

Отримаємо:

```
['a,', 'b,', 'c,', 'd,', 'e']
```

Але в будь-якому випадку навіть при виклику функції *split()* без аргументів, все одно потрібно додавати круглі дужки – саме так Python дізнається, що викликається функція.

### **Об'єднання рядків за допомогою функції *join()***

Функція *join()* є протилежністю функції *split()*: об'єднує список рядків в один рядок.

Для виклику функції спочатку вказується рядок, який об'єднує інші, а потім – список рядків для об'єднання:

### **рядок.*join*(список)**

Для того щоб об'єднати список рядків *lines*, розділивши їх символами нового рядка, потрібно написати *'\n'.join(lines)*.

```
q = ['abcdefgh', 'ijklmnopqr', 'stuvwxyz']
string = ".join(q)           # Об'єднання 3 послідовностей
                             літер без розділення

print(string)
string = ', '.join(q)       # Об'єднання 3 послідовностей
                             літер з розділенням їх комами та
                             пробілом

print(string)
```

Результатом запуску даного коду буде:

```
abcdefghijklmnoqrstuvwxyz
abcdefgh, ijklmnopqr, stuvwxyz
```

### **Регістр і вирівнювання**

В Python є велика множина строкових методів (можуть бути використані з будь-яким об'єктом *str*) і модуль *string*, що містить корисні визначення.

```
string = "abcdefghijklmnoqrstuvwxyz"
string = ".abcdefghijklhijklmnopqrs tuvxyz..."
```

```

q = string.strip('.')          # Видалення символів '.' з обох
кінців рядка:
print(q)

q = string.capitalize()      # Перше слово з великої літери
print(q)

q = string.title()          # Всі слова з великої літери
print(q)

q = string.upper()          # Всі слова великими літерами
print(q)

q = string.lower()          # Всі слова маленькими літерами
print(q)

q = string.swapcase()        # Зміна регістру літер
print(q)

```

Результатом запуску даного коду буде:

```

abcdefghi jklmnopqrs tuvxyz
.abcdefghi jklmnopqrs tuvxyz...
.Abcdefghi Jklmnopqrs Tuvxyz...
.ABCDEFGHI JKLMNOPQRS TUVWXYZ...
.abcdefghi jklmnopqrs tuvxyz...
.ABCDEFGHI JKLMNOPQRS TUVWXYZ...

```

### ***Форматування рядків***

Python дозволяє виконати вирівнювання рядків [5, 8].

```

string = ".abcdefghi jklmnopqrs tuvxyz..."
q=string.center(60)      # Рядок вирівнюється всередині заданої
                          кількості пробілів (30) по центру
print(q)

q=string.ljust(60)       # Рядок вирівнюється по лівому краю
print(q)

```

```
q=string.rjust(60)           # Рядок вирівнюється по
                             правому краю
print(q)
```

Результатом запуску даного коду буде:

```
.abcdefghijklmnopqrs tuvwxyz...
.abcdefghijklmnopqrs tuvwxyz...
.abcdefghijklmnopqrs tuvwxyz...
```

Python дозволяє *інтерполювати дані в рядки* – розмістити значення всередині рядків, – застосовуючи різні формати. Можна використовувати цю можливість, щоб створювати звіти та інші документи, яким необхідно зробити певний зовнішній вигляд.

Python пропонує два способи форматування рядків.

### **Форматування рядків з використанням символу %**

Форматування рядків з використанням символу % має форму:

**рядок % дані**

Усередині рядка знаходяться інтерполяційні послідовності. У табл. 2.5 показано, що найпростіша послідовність – це символ %, за яким слідує буква, що представляє тип даних, який повинен бути відформатований.

Таблиця 2.5. Типи перетворення

%s	Рядок
%d	Ціле число в десятковій системі числення
%x	Ціле число в шістнадцятковій системі числення
%o	Ціле число в вісімковій системі числення
%f	Число з плаваючою крапкою в десятковій системі числення
%e	Число з плаваючою крапкою в шістнадцятковій системі числення
%g	Число з плаваючою крапкою у вісімковій системі числення
%%	Символ %

Послідовність %s всередині рядка означає, що в неї потрібно інтерполювати рядок. Кількість використаних символів %

повинно збігатися з кількістю об'єктів, які розташовуються після %.

Ціле число:

```
q = '%s' % 42  
print(q)
```

```
q = '%d' % 42  
print(q)
```

```
q = '%x' % 42  
print(q)
```

```
q = '%o' % 42  
print(q)
```

Отримаємо:

```
42  
42  
2a  
52
```

Число з плаваючою крапкою:

```
q = '%s' % 7.03  
print(q)
```

```
q = '%f' % 7.03  
print(q)
```

```
q = '%e' % 7.03  
print(q)
```

```
q = '%g' % 7.03  
print(q)
```

Отримаємо:

```
7.03  
7.030000  
7.030000e+00
```

### 7.03

Ціле число і символ %:

```
q = '%d%%' % 100  
print(q)
```

Отримаємо:

```
100%
```

Інтерполяція деяких рядків і цілих чисел:

```
breed = 'British Shorthair'  
cat = 'Lola'  
weight = 4  
  
q = "My favorite cat breed is %s" % breed  
print(q)  
  
q = "My cat %s weighs %s kg" % (cat, weight)  
print(q)
```

Результатом запуску даного коду буде:

```
My favorite cat breed is British Shorthair  
My cat Lola weighs 4 kg
```

Один об'єкт на зразок *breed* розташовується відразу після символу %. Якщо таких об'єктів кілька, вони повинні бути згруповані в кортеж (потрібно оточити їх дужками і розділити комами) на зразок (*cat, weight*).

Незважаючи на те що змінна *weight* цілочисельна, послідовність *%s* всередині рядка перетворює її в рядок.

Можна додати інші значення між % і визначенням типу, щоб вказати мінімальну і максимальну ширину, вирівнювання і заповнення символами.

```
n = 42  
f = 7.03  
s = 'string'  
  
q = '%d %f %s' % (n, f, s)
```

```
print(q)
```

Отримаємо:

```
'42 7.030000 string'
```

Можна встановити мінімальну довжину поля для кожної змінної і вирівняти їх по правому (лівому) краю, заповнюючи невикористане місце пробілами:

```
n = 42
f = 7.03
s = 'string'
q = '% 10d % 10f % 10s' % (n, f, s)      # Вирівнювання по
                                         # правому краю,
                                         # мінімальна довжина
                                         # поля = 10 знаків

print(q)

q = '%-10d %-10f %-10s' % (n, f, s)     # Вирівнювання по
                                         # лівому краю, мінімальна
                                         # довжина поля = 10
                                         # знаків

print(q)
```

Отримаємо:

```
    42  7.030000  string
42     7.030000  string
```

Можна вказати довжину поля та максимальну кількість символів (вирівнювання по правому краю). Таке налаштування обрізає рядок і обмежує число з плаваючою точкою чотирма цифрами після десяткової коми:

```
n = 42
f = 7.03
s = 'string'

q = '% 10.4d % 10.4f % 10.4s' % (n, f, s)

print(q)
```

```
q = '%.4d %.4f %.4s' % (n, f, s)
print(q)
```

Отримаємо:

```
0042 7.0300 stri
0042 7.0300 stri
```

### **Форматування за допомогою символів {} і функції format**

В Python 3 рекомендується застосовувати новий стиль форматування за допомогою методу *format()*, що має наступний синтаксис:

**рядок\_спеціального\_формату.format(\*args, \*\*kwargs)**

У параметрі *рядок\_спеціального\_формату* всередині символів {} можуть бути вказані деякі специфікатори.

Всі символи, розташовані поза фігурних дужок, виводяться без перетворень. Якщо всередині рядка необхідно використовувати символи {}, то ці символи слід подвоїти, інакше збуджується виняток *ValueError*.

```
n = 42
f = 7.03
s = 'string'
q = '{} {} {}'.format(n, f, s)

print(q)
```

Отримаємо:

```
42 7.03 string
```

Аргументи старого стилю потрібно надавати в порядку появи їх наповнювачів з символами % в оригінальній рядку. За допомогою нового стилю можна вказувати будь-який порядок:

```
n = 42
f = 7.03
s = 'string'
```

```
q = '{2} {0} {1}'.format(f, s, n)
print(q)
```

Отримаємо:

```
42 7.03 string
```

Значення *0* відноситься до першого аргументу, *f*, *1* відноситься до рядка *s*, а *2* – до останнього аргументу, цілого числа *n*.

Аргументи можуть бути словником або іменованими аргументами, а специфікатори можуть включати їх імена:

```
q = '{n} {f} {s}'.format(n=42, f=7.03, s='string')
print(q)
```

Отримаємо:

```
42 7.03 string
```

Старий стиль дозволяє вказати специфікатор типу після символу %, а новий стиль – після .:

```
n = 42
f = 7.03
s = 'string'
```

```
q = '{0:d} {1:f} {2:s}'.format(n, f, s)
print(q)
```

# Для іменованих аргументів

```
q = '{n:d} {f:f} {s:s}'.format(n=42, f=7.03, s='string')
print(q)
```

Отримаємо:

```
42 7.030000 string
42 7.030000 string
```

Інші можливості (мінімальна довжина поля, максимальна ширина символів, зміщення і т.д.) також підтримуються.

```
n = 42
f = 7.03
s = 'string'
```



```

# Мінімальна довжина поля 10
q = '{0:10d} {1:10f} {2:10s}'.format(n, f, s)
print(q)

# Вирівнювання по правому краю
q = '{0:>10d} {1:>10f} {2:>10s}'.format(n, f, s)
print(q)

# Вирівнювання по лівому краю
q = '{0:<10d} {1:<10f} {2:<10s}'.format(n, f, s)
print(q)

# Вирівнювання по центру
q = '{0:^10d} {1:^10f} {2:^10s}'.format(n, f, s)
print(q)

```

Результатом запуску даного коду буде:

```

42 7.030000 string
42 7.030000 string
42 7.030000 string
42 7.030000 string

```

Значення точності (після десяткової коми) означає кількість цифр після десяткової коми для дробових чисел і максимальне число символів рядка, але не можна використовувати його для цілих чисел:

```

n = 42
f = 7.03
s = 'string'

# Без використання для цілих чисел
q = '{0:>10d} {1:>10.4f} {2:>10.4s}'.format(n, f, s)

print(q)

# Використано для цілих чисел
q = '{0:>10.4d} {1:>10.4f} {2:10.4s}'.format(n, f, s)

```

```
print(q)
```

Отримаємо:

```
42 7.0300 stri
```

Traceback (most recent call last):

File "G:\lab.py", line 5, in <module>

```
q = '{0:>10.4d} {1:>10.4f} {2:10.4s}'.format(n, f, s)
```

ValueError: Precision not allowed in integer format specifier

Якщо необхідно заповнити поле виведення чимось крім пробілів, можна розмістити необхідний символ відразу після двокрапки, але перед символами вирівнювання (<, >, ^) або специфікатором ширини:

```
q = '{0:!20s}'.format('ВЕЛИКІ ЛІТЕРИ')
```

```
print(q)
```

Отримаємо:

```
!!!ВЕЛИКІ ЛІТЕРИ!!!!
```

### ***Заміна символів***

Можна використовувати функцію *replace()* для того, щоб замінити один підрядок іншим. В функцію передається старий підрядок, новий підрядок і кількість включень старого підрядка, яку потрібно замінити. Якщо опустити останній аргумент, будуть замінені всі включення.

```
x='a a aaa a b ba ba ab cb bc'
```

```
q = x.replace('a', 'y')
```

```
print(q)
```

```
q = x.replace('a', 'y', 5)
```

# Заміна 5 входжень

```
print(q)
```

Отримаємо:

```
y y ууу y b by by yb cb bc
```

```
y y ууу a b ba ba ab cb bc
```

Але треба бути обережним, оскільки можна виконати заміну символів в середині слів.

## 2.9. Складні структури даних. Списки

Масив – набір фіксованої кількості елементів, що розміщені в пам'яті комп'ютера безпосередньо один за одним, а доступ до них здійснюється за індексом (номер даного елементу в масиві).

В Python для реалізації масиву використовуються списки. **Список** – тип даних, що представляє собою послідовність певних значень, що можуть повторюватись. Але на відміну від масиву – кількість елементів у списку може бути довільною.

**Списки** – гетерогенна, змінювана структура даних, що може містити елементи різних типів, що перераховані через кому та заключені в квадратні дужки. Це дозволяє створювати структури будь-якої складності і глибини.

Списки служать для того, щоб зберігати об'єкти в певному порядку, особливо якщо порядок або вміст можуть змінюватися. Можна змінювати список, додати в нього нові елементи, а також видалити або перезаписати існуючі. Можна змінити кількість елементів у списку, а також самі елементи. Одне і те ж значення може зустрічатися в списку кілька разів.

Список є об'єктом, тому може бути присвоєний змінній.

```
int_list=[1, 2, 5, 8]
```

Отримаємо:

```
[1, 2, 5, 8]
```

Список можна створити з нуля або більше елементів, розділених комами і вкладених у квадратні дужки:

```
empty_list = [ ]
number_list = [1, 2, 3, 4, 5]
week_days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday']

print(empty_list)
print(number_list)
print(week_days)
```

Отримаємо:

```
[]  
[1, 2, 3, 4, 5]  
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Крім того, за допомогою функції *list()* можна створити порожній список:

```
another_empty_list = list()  
  
print (another_empty_list)
```

Отримаємо:

```
[]
```

Функція *list()* перетворює інші типи даних в списки.

```
# Рядок перетвориться в список, що складається з  
# односимвольних рядків  
q = list('cat')  
print(q)
```

Отримаємо:

```
['c', 'a', 't']
```

### ***Звернення до елемента***

Список містить різні дані, звертатися до яких можна через ім'я списку та вказавши зміщення необхідного елемента:

```
letters_list = ['a', 'b', 'c']  
  
print(letters_list)           # Виведення всього списку  
print(letters_list[0])       # Виведення першого елемента  
                               списку  
print(letters_list[1])       # Виведення другого елемента  
                               списку  
print(letters_list[2])       # Виведення третього елемента  
                               списку  
print(letters_list[-1])      # Виведення останнього елемента  
                               списку
```

```
print(letters_list[-2])      # Виведення передостаннього  
                             елементу списку
```

Результатом запуску даного коду буде:

```
a  
b  
c  
c  
b
```

Зсув повинен бути коректним значенням для списку – воно являє собою позицію, на якій розташовується присвоєне раніше значення. Якщо вказати позицію, яка знаходиться перед списком або після нього, буде згенеровано виняток (помилка).

```
letters_list = ['a', 'b', 'c']  
print(letters_list[3])
```

Отримаємо:

```
Traceback (most recent call last):  
  File "G:\lab.py", line 9, in <module>  
    print(letters_list[3])  
IndexError: list index out of range
```

За аналогією з отриманням значення списку за допомогою його зміщення можна змінити це значення:

```
letters_list = ['a', 'b', 'c']  
print (letters_list)
```

```
letters_list[2] = 'C'  
print (letters_list)
```

Отримаємо:

```
['a', 'b', 'c']  
['a', 'b', 'C']
```

### ***Отримання елементів за допомогою діапазону зсувів***

Можна отримати зі списку підпоследовність, використавши зріз списку:

```

letters_list = ['a', 'b', 'c', 'd', 'e']

print(letters_list[0:2])      # Виведення елементів починаючи
                              з 1-го до 2-го
print(letters_list[::2])     # Кожен непарний елемент
print(letters_list[::-2])    # Всі елем. з останнього зі
                              зміщенням вліво на 2:
print(letters_list[::-1])    # Інверсія списку

```

Отримаємо:

```

['a', 'b']
['a', 'c', 'e']
['e', 'c', 'a']
['e', 'd', 'c', 'b', 'a']

```

### ***Методи списків***

Для додавання елементів в кінець списку – використовують метод ***append()***.

```

letters_list = ['a', 'b', 'c', 'd', 'e']
print(letters_list)

letters_list.append('f')

print(letters_list)

```

Результатом запуску даного коду буде:

```

['a', 'b', 'c', 'd', 'e']
['a', 'b', 'c', 'd', 'e', 'f']

```

Можна об'єднати один список з іншим за допомогою методу ***extend()***.

```

letters_list = ['a', 'b', 'c', 'd', 'e']
others_list = ['g', 'h', 'i']
print(letters_list)
print(others_list)

letters_list.extend(others_list)
print(letters_list)

```

Результатом запуску даного коду буде:

```
['a', 'b', 'c', 'd', 'e']  
['g', 'h', 'i']  
['a', 'b', 'c', 'd', 'e', 'g', 'h', 'i']
```

Можна також використовувати оператор `+=`:

```
letters_list = ['a', 'b', 'c', 'd', 'e']  
others_list = ['g', 'h', 'i']  
print(letters_list)  
print(others_list)  
  
letters_list += others_list  
print(letters_list)
```

Результат буде аналогічний.

При використанні методу ***append()***, список *others\_list* був би доданий як один елемент списку, замість того щоб об'єднати його елементи зі списком *letters\_list*:

```
letters_list = ['a', 'b', 'c', 'd', 'e']  
others_list = ['g', 'h', 'i']  
print(letters_list)  
print(others_list)  
  
letters_list.append(others_list)  
print(letters_list)
```

Отримаємо:

```
['a', 'b', 'c', 'd', 'e', 'f', ['g', 'h', 'i']]
```

Функція ***append()*** додає елементи тільки в кінець списку. Коли потрібно додати елемент в задану позицію, використовується функція ***insert()***. Якщо вказати позицію 0, елемент буде додано в початок списку. Якщо позиція знаходиться за межами списку, елемент буде додано в кінець списку, як і у випадку з функцією ***append()***, виняток не буде згенеровано:

```
letters_list = ['a', 'b', 'c', 'd', 'e']  
print(letters_list)
```

```
letters_list.insert(6, 'g')
print(letters_list)
```

```
letters_list.insert(10, 'k')
print(letters_list)
```

```
letters_list.insert(2, 'm')
print(letters_list)
```

Результатом запуску даного коду буде:

```
['a', 'b', 'c', 'd', 'e']
['a', 'b', 'c', 'd', 'e', 'g']
['a', 'b', 'c', 'd', 'e', 'g', 'k']
['a', 'b', 'm', 'c', 'd', 'e', 'g', 'k']
```

### ***Видалення заданого елемента***

Коли видаляється заданий елемент, всі інші елементи, які йдуть слідом за ним, зміщуються, щоб зайняти місце видаленого елемента, а довжина списку зменшується на одиницю. Один із варіантів видалення елемента є застосування інструкції ***del***:

```
letters_list = ['a', 'b', 'c', 'd', 'e']
print(letters_list)

del letters_list [2]                # Видалення другого
                                    елементу

print(letters_list)
print(letters_list [2])

del letters_list [-1]              # Видалення останнього
                                    елементу

print(letters_list)
print(letters_list [-1])
```

Отримаємо:

```
['a', 'b', 'c', 'd', 'e']
['a', 'b', 'd', 'e']
d
['a', 'b', 'd']
```



```
d
```

Якщо точно не відомо або все одно, в якій позиції знаходиться елемент, використовується функція *remove()*, щоб видалити його за значенням:

```
letters_list=['a', 'b', 'c', 'd', 'e']
print(letters_list)

letters_list.remove('b')
print(letters_list)
print(letters_list [1])
```

Отримаємо:

```
['a', 'b', 'c', 'd', 'e']
['a', 'c', 'd', 'e']
c
```

За допомогою функції *pop()* можна отримати елемент зі списку і в той же час видалити його. Якщо викликати функцію *pop()* і вказати зсув, вона поверне елемент, що знаходиться в заданій позиції. Якщо аргумент не вказано – буде використано значення -1. Так, виклик *pop(0)* поверне головний (початковий) елемент списку, а виклик *pop()* або *pop(-1)* – кінцевий елемент:

```
letters_list=['a', 'b', 'c', 'd', 'e']

print(letters_list)

print(letters_list.pop(1))    # Повертаємо значення видаленого
                              елементу

print(letters_list)
print(letters_list [1])
```

Отримаємо:

```
['a', 'b', 'c', 'd', 'e']
b
['a', 'c', 'd']
c
```

### ***Визначення зміщення елемента по значенню***

Щоб визначити зміщення елемента в списку по його значенню, використовується функція *index()*:

```
letters_list = ['a', 'b', 'c', 'd', 'e']  
  
print(letters_list)  
  
print(letters_list.index('d'))
```

Отримаємо:

```
['a', 'b', 'c', 'd', 'e']  
3
```

### ***Перевірка на наявність елемента в списку за допомогою оператора in***

В Python наявність елемента в списку перевіряється за допомогою оператора *in*:

```
letters_list = ['a', 'b', 'c', 'd', 'e']  
  
print("'d' in letters_list - ", 'd' in letters_list)  
print("'d' in letters_list - ", 'b' in letters_list)
```

Отримаємо:

```
'd' in letters_list - True  
'd' in letters_list - True
```

Одне і те ж значення може зустрітися більше одного разу. До тих пір поки воно знаходиться в списку хоча б в одному екземплярі, оператор *in* буде повертати значення *True*.

Щоб визначити, скільки разів якийсь значення зустрічається в списку, використовується функція *count()*:

```
letters_list = ['a', 'b', 'c', 'd', 'e']  
  
print(letters_list.count('b'))  
print(letters_list.count('o'))
```

Отримаємо:

```
0
2
```

### ***Зміна порядку елементів за допомогою функції `sort()`***

Щоб змінювати порядок елементів за їх значенням, а не за зсувам в Python є дві функції:

- функція списку `sort()`, яка сортує сам список;
- функція `sorted()`, яка повертає відсортовану копію списку.

Якщо елементи списку є числами, вони за замовчуванням сортуються по зростанню. Якщо рядками, то сортуються в алфавітному порядку:

```
names = ['Alex', 'Olga', 'Helen']
print(names)
sorted_names = sorted(names)
print(sorted_names)
print(names)
```

`sorted_names` – це копія, її створення не змінило оригінальний список:

```
['Alex', 'Olga', 'Helen']
['Alex', 'Helen', 'Olga']
['Alex', 'Olga', 'Helen']
```

Виклик функції списку `sort()` для `names` змінить цей список:

```
names = ['Alex', 'Olga', 'Helen']
print(names)

names.sort()
print(names)
```

Отримаємо:

```
['Alex', 'Olga', 'Helen']
['Alex', 'Helen', 'Olga']
```

Якщо всі елементи списку одного типу, функція `sort()` відпрацює коректно. Іноді можна навіть змішати типи – наприклад, цілі числа і числа з плаваючою точкою, – оскільки в рамках виразів вони конвертуються автоматично.

За замовчуванням список сортується по зростанню, але можна додати аргумент *reverse = True*, щоб впорядкувати список за зменшенням:

```
numbers = [2, 1, 4.0, 3]
print(numbers)

numbers.sort()
print(numbers)

# Щоб впорядкувати список у спадному порядку
numbers.sort(reverse=True)
print(numbers)

# Визначаємо кількість елементів списку
print("Кількість елементів списку numbers: ", len(numbers))
```

Отримаємо:

```
[2, 1, 4.0, 3]
[1, 2, 3, 4.0]
[4.0, 3, 2, 1]
Кількість елементів списку numbers: 4
```

***Присвоєння за допомогою оператора =, копіювання за допомогою функції copy()***

Якщо ви привласнюєте один список більш ніж одній змінній, зміна списку в одному місці спричинить за собою його зміну в інших:

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

b = a
print(b)

a[0] = 5

print(a)
print(b)
```

Отримаємо:

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

Змінна *b* просто посилається на той же список об'єктів, що і *a*, тому, незалежно від того, за допомогою якого імені ми змінюємо вміст списку, зміна відіб'ється на обох змінних:

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

b = a
print(b)

b[0] = 6

print(a)
print(b)
```

Отримаємо:

```
[1, 2, 3]
[1, 2, 3]
[6, 2, 3]
[6, 2, 3]
```

Можна скопіювати значення в незалежний новий список за допомогою одного з таких методів:

- функція *copy()*;
- функція перетворення *list()*;
- розбиття списку *[:]*.

Оригінальний список знову буде присвоєно змінній *a*. Створимо *b* за допомогою функції списку *copy ()*, *c* – за допомогою функції перетворення *list ()*, а *d* – за допомогою розбиття списку:

```
a = [1, 2, 3]

b = a.copy()
```

```
c = list(a)
d = a[:]

print("a: ", a)
print("b: ", b)
print("c: ", c)
print("d: ", d)

a[0] = 5

print("Після змін:")
print("a: ", a)
print("b: ", b)
print("c: ", c)
print("d: ", d)
```

*b*, *c* і *d* є копіями *a* – це нові об’єкти, що мають свої значення, не пов’язані з оригінальним списком об’єктів [1, 2, 3], на який посилається *a*. Зміна *a* не вплине на копії *b*, *c* і *d*:

```
a: [1, 2, 3]
b: [1, 2, 3]
c: [1, 2, 3]
d: [1, 2, 3]
Після змін:
a: [5, 2, 3]
b: [1, 2, 3]
c: [1, 2, 3]
d: [1, 2, 3]
```

## 2.10. Складні структури даних. Кортежі

Кортежі, як і списки, є послідовностями довільних елементів [5 – 13]. На відміну від списків кортежі незмінні, коли привласнюється кортежу елемент, він "запікається" і більше не змінюється.

Всі операції над списками, що не змінюють список (додавання, множення на число, функції *index()* і *count()* і деякі

інші операції) можна застосовувати до кортежів. Можна також по-різному змінювати елементи місцями і так далі.

### ***Створення кортежів за допомогою оператора ()***

Щоб створити порожній кортеж використовується оператор ():

```
empty_tuple = ()  
print(empty_tuple)
```

Отримаємо:

```
()
```

Щоб створити кортеж, що містить один елемент або більше, необхідно ставити після кожного елемента кому. У випадку з одним елементом кома повинна обов'язково стояти після цього єдиного елемента.

```
one_name = 'Alex',  
print(one_name)
```

Отримаємо:

```
('Alex',)
```

Якщо у кортежі більше одного елемента, ставиться кома після кожного з них, крім останнього:

```
name_tuple = 'Alex', 'Helen', 'Olga'  
print(name_tuple)
```

Отримаємо:

```
('Alex', 'Helen', 'Olga')
```

При створенні кортежу можна заключити елементи у круглі дужки, що дозволяє зробити кортежі більш помітними:

```
name_tuple = ('Alex', 'Helen', 'Olga')  
print(name_tuple)
```

Отримаємо:

```
('Alex', 'Helen', 'Olga')
```

Кортежі дозволяють привласнити кілька змінних за один раз:

```
name_tuple = ('Alex', 'Helen', 'Olga')
a, b, c = name_tuple
```

```
print (a)
print (b)
print (c)
```

Отримаємо:

```
'Alex'
'Helen'
'Olga'
```

Іноді це називається розпакуванням кортежу.

Можна використовувати кортежі, щоб обміняти значення за допомогою одного виразу, без застосування тимчасової змінної:

```
a = 1
b = 2
a, b = b, a
```

```
print (a)
print (b)
```

Отримаємо:

```
2
1
```

Функція перетворення *tuple()* створює кортежі з інших об'єктів:

```
name_list = ['Alex', 'Helen', 'Olga']
q = tuple(name_list)
print (q)
```

Отримаємо:

```
('Alex', 'Helen', 'Olga')
```

Можна використовувати кортежі замість списків, але вони мають менше можливостей, оскільки кортеж не може бути змінений після створення. Переваги кортежів:



- Захист від дурня, оскільки кортеж захищений від змін, як навмисних (що погано), так і випадкових (що добре).
- Кортежі займають менше місця.
- Можна використовувати кортежі як ключі словника.
- Іменовані кортежі.
- Аргументи функції передаються як кортежі.

## 2.11. Складні структури даних. Словники

Словник дуже схожий на список, але порядок елементів в ньому не має значення, і вони вибираються не за допомогою зміщення. Замість цього для кожного значення вказується пов'язаний з ним унікальний ключ. Таким ключем може бути об'єкт одного з незмінних типів: рядок, булева змінна, ціле число, число з плаваючою точкою, кортеж і іншими об'єктами. Елементи словника можуть містити об'єкти довільного типу даних і мати необмежений рівень вкладеності. Елементи в словниках розташовуються в довільному порядку [5 – 13].

Словники можна змінювати – це означає, що можна додати, видалити і змінити їх елементи, які мають вигляд "ключ – значення".

### *Створення словника за допомогою {}*

Щоб створити словник, необхідно заключити в фігурні дужки (*{}*) розділені комами пари **ключ: значення**.

```
dict_1 = {'a': 1, 'b': 2}
print(dict_1)
```

Отримаємо:

```
{'b': 2, 'a': 1}
```

Уведення імені словника в інтерактивний інтерпретатор виведе всі його ключі і значення (вміст словника).

Можна використовувати функцію *dict()*, щоб створити порожній словник, якщо не вказати параметри функції.

```
empty_dict = dict()
print(empty_dict)
```

Отримаємо:

```
{}
```

Можна використовувати функцію *dict()*, щоб перетворювати послідовності з двох значень в словники.

```
dict_1 = dict({"a": 1, "b": 2})           # Словник
dict_2 = dict(("a", 1), ("b", 2))       # Список кортежей
dict_3 = dict(["a", 1], ["b", 2])       # Список списків

print("dictionary_1: ", dict_1)
print("dictionary_2: ", dict_2)
print("dictionary_3: ", dict_3)
```

Результатом запуску даного коду буде:

```
dictionary_1: {'b': 2, 'a': 1}
dictionary_2: {'b': 2, 'a': 1}
dictionary_3: {'b': 2, 'a': 1}
```

Можна використовувати будь-яку послідовність, що містить послідовності, які складаються з двох елементів. Розглянемо інші приклади.

Список, що містить двосимвольні рядки:

```
dict_1 = dict(['ab', 'cd', 'ef'])
print("dictionary: ", dict_1)
```

Отримаємо:

```
dictionary: {'e': 'f', 'c': 'd', 'a': 'b'}
```

Об'єднати два списки в список кортежів дозволяє функція *zip()*:

```
x = ["a", "b"]           # Список з ключами
y = [1, 2]               # Список зі значеннями

# Створення словника зі списку кортежів [('a', 1), ('b', 2)]
dict_1 = dict(zip(x, y))

print("dictionary: ", dict_1)
```

Отримаємо:

```
dictionary: {'a': 1, 'b': 2}
```

Звернення до елементів словника здійснюється за допомогою квадратних дужок, в яких вказується ключ. Як ключ можна вказати незмінний об'єкт, наприклад, число, рядок або кортеж.

```
dict_1 = {1: "a", 3: "c", 4: "d"}
```

```
print (dict_1[1])  
print (dict_1[3])
```

Отримаємо:

```
'a'  
'c'
```

Якщо елемент, відповідний вказаному ключу, відсутній в словнику, то збуджується виняток *KeyError*:

```
dict_1 = {1: "a", 3: "c", 4: "d"}  
print (dict_1[2])
```

Отримаємо:

```
Traceback (most recent call last):  
  File "G:\lab.py", line 2, in <module>  
    print (dict_1[2])  
KeyError: 2
```

Є два способи уникнути виникнення цього винятку.

Перший з них – перевірити, чи є заданий ключ, за допомогою ключового слова *in* у словнику.

Другий спосіб – використати спеціальну функцію словника *get()*.

Вказується словник, ключ і необов'язкове значення. Якщо ключ існує, буде отримано пов'язане з ним значення:

```
dict_1 = {1: "a", 3: "c", 4: "d"}  
q = dict_1.get(1)  
print(q)
```

Отримаємо:

```
'a'
```

Якщо такого ключа немає, буде отримано необов'язкове значення, якщо воно було вказане:

```
dict_1 = {1: "a", 3: "c", 4: "d"}
q = dict_1.get('2', 'Not a dict')

print(q)
```

Отримаємо:

```
'Not a dict'
```

В іншому випадку буде повернуто об'єкт `None` (інтерактивний інтерпретатор не виведе нічого):

```
dict_1 = {1: "a", 3: "c", 4: "d"}
q = dict_1.get('2')

print(q)
```

Отримаємо:

```
None
```

### ***Перевірка наявності ключа***

Щоб дізнатися, чи міститься в словнику якийсь ключ, використовується ключове слово *in*. Якщо ключ знайдений, то повертається значення *True*, в іншому випадку – *False*.

```
dict_1 = {1: "a", 3: "c", 4: "d"}

print(1 in dict_1)
print(2 in dict_1)
```

Отримаємо:

```
True
False
```

### ***Додавання або зміна елемента***

Оскільки словники відносяться до змінюваних типів даних, то можна додати або змінити елемент по ключу.

Додати елемент в словник досить легко. Потрібно просто звернутися до елемента по його ключу і привласнити йому значення. Якщо ключ вже існує в словнику, наявне значення буде

замінено новим. Якщо ключ новий, він і вказане значення будуть додані в словник.

```
dict_1 = {1: "a", 3: "c", 4: "d"}
dict_1[2] = "c" # Додавання нового елемента
print(dict_1)

dict_1[2] = "b" # Зміна елемента по ключу
print(dict_1)
```

Отримаємо:

```
{1: 'a', 2: 'c', 3: 'c', 4: 'd'}
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

На відміну від списків Python не згенерує виняток під час привласнення нового елемента, якщо вказати, що цей індекс знаходиться поза існуючого діапазону. Ключі в словнику повинні бути унікальними. Якщо застосовувати ключ більш ніж один раз, останнє значення буде мати найвищий пріоритет і саме воно займе місце у словнику:

```
dict_1 = {1: "a", 3: "c", 4: "d"}
dict_1[5] = "e"
print(dict_1)

dict_1[5] = "f"
print(dict_1)
```

Отримаємо:

```
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e'}
{1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'f'}
```

### **Методи словників**

***update()*** – додає елементи в словник. Метод змінює поточний словник і нічого не повертає.

Якщо елемент із зазначеним ключем вже присутній в словнику, то його значення буде перезаписано.

```
dict_2 = {"a": 1, "b": 2}
```

```
dict_2.update(c=3, d=4)
print(dict_2)

dict_2.update({"c": 10, "d": 20})           # Словник
print(dict_2)

dict_2.update([("d", 80), ("e", 6)])       # Список кортежей
print(dict_2)

dict_2.update([["a", "str"], ["i", "t"]])  # Список списків
print(dict_2)
```

Результатом запуску даного коду буде:

```
{'a': 1, 'b': 2, 'd': 4, 'c': 3}
{'b': 2, 'c': 10, 'a': 1, 'd': 20}
{'e': 6, 'b': 2, 'c': 10, 'a': 1, 'd': 80}
{'e': 6, 'b': 2, 'c': 10, 'a': 'str', 'd': 80, 'i': 't'}
```

Можна використовувати функцію *update()*, щоб скопіювати ключі і значення з одного словника в іншій.

### **Видалення елементів**

Видалити елемент зі словника можна за допомогою інструкції *del*:

```
dict_2 = {"a": 1, "b": 2}
print(dict_2)

del dict_2 ["b"]           # Видаляємо елемент з ключем "b"
print(dict_2)
```

Отримаємо:

```
{"a": 1, "b": 2}
{'a': 1}
```

Щоб видалити всі ключі і значення зі словника, слід використовувати функцію *clear()* або просто привласнити порожній словник заданому імені:

```
dict_2 = {"a": 1, "b": 2}
dict_2.clear() # Видаляємо всі елементи

print(dict_2)
```

Отримаємо:

```
{}
```

Скориставшись функцією *keys()* можна отримати всі ключі словника.

```
dict_2 = {"a": 1, "b": 2}
print(dict_2 )

print(dict_2.keys())
print(list(dict_2.keys()))
```

Результатом запуску даного коду буде:

```
{'a': 1, 'b': 2}
dict_keys(['a', 'b'])
['a', 'b']
```

Щоб отримати всі значення словника, використовується функція *values()*:

```
dict_2 = {"a": 1, "b": 2}

print(dict_2 )
print(dict_2.values())
print(list(dict_2. values ()))
```

Результатом запуску даного коду буде:

```
{'b': 2, 'a': 1}
dict_values([2, 1])
[2, 1]
```

Щоб отримати всі пари "ключ – значення" із словника, використовується функція *items()*:

```
dict_2 = {"a": 1, "b": 2}
print(dict_2 )
```

```
print(dict_2.items())
print(list(dict_2.items()))
```

Результатом запуску даного коду буде:

```
{'b': 2, 'a': 1}
dict_items([('b', 2), ('a', 1)])
[('b', 2), ('a', 1)]
```

Кожна пара буде повернена як кортеж.

***Надання значення за допомогою оператора =, копіювання їх за допомогою функції copy()***

Як і у випадку зі списками, якщо потрібно внести в словник зміну, вона відіб'ється для всіх імен, які посилаються на нього. Щоб скопіювати ключі і значення з одного словника в іншій і уникнути цього, можна скористатися функцією *copy()*:

```
dict_2 = {"a": 1, "b": 2}

dict_copy = dict_2
dict_copy_2 = dict_2.copy()

print('dict_2', dict_2)
print('dict_copy', dict_copy)
print('dict_copy_2', dict_copy_2)

dict_2 ['c'] = 3
print('dict_2 після змін', dict_2)
print('dict_copy після змін', dict_copy)
print('dict_copy_2 після змін', dict_copy_2)
```

Результатом запуску даного коду буде:

```
dict_2 {'a': 1, 'b': 2}
dict_copy {'a': 1, 'b': 2}
dict_copy_2 {'a': 1, 'b': 2}
dict_2 після змін {'a': 1, 'c': 3, 'b': 2}
dict_copy після змін {'a': 1, 'c': 3, 'b': 2}
dict_copy_2 після змін {'a': 1, 'b': 2}
```

**Завдання на комп'ютерний практикум**



- Змінній `var_int` надайте значення 10, `var_float` – значення 8.4, `var_str` – "No".
- Змініть значення, збережене в змінній `var_int`, збільшивши його в 3.5 рази, результат зв'яжіть зі змінною `big_int`.
- Змініть значення, збережене в змінній `var_float`, зменшивши його на одиницю, результат зв'яжіть з тією ж змінною.
- Змініть значення змінної `var_str` на "NoNoYesYesYes". При формуванні нового значення використовуйте операції конкатенації (+) і повторення рядка (\*).

### Запитання для самоконтролю

1. Які типи даних ви знаєте? Опишіть їх.
2. Чи можна перетворити дробове число в ціле? Ціле в дробове? У яких випадках можна рядок перетворити в число?
3. Наведіть приклади операцій. Для чого призначена операція присвоєння?
4. Які існують правила і рекомендації для іменування змінних?
5. Наведіть приклади використання функцій оброблення символів.
6. Які функції існують для уведення і виведення символів?
7. Як отримати довжину рядка?
8. Що таке кортеж?
9. Що таке словник?
10. Які методи списків ви знаєте?

## РОЗДІЛ 3. АЛГОРИТМІЧНІ СТРУКТУРИ В МОВІ PYTHON

### 3.1. Основні алгоритмічні структури

Основними алгоритмічними структурами є: слідування, розгалуження, цикл.

- **Слідування** – команди виконуються послідовно одна за іншою.

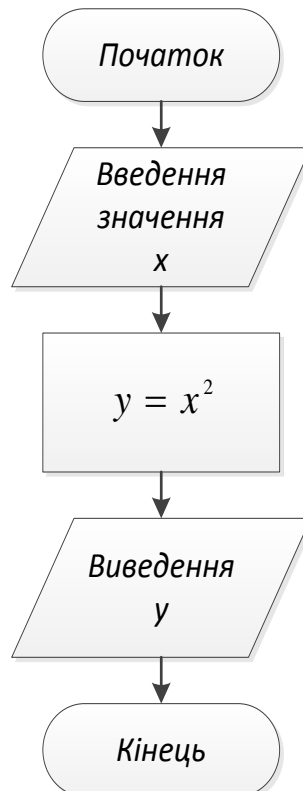


Рис.3.1. Приклад блок-схеми реалізації лінійного алгоритму

- **Розгалуження** – алгоритм, що містить хоча б одну умову в результаті перевірки якої може виконуватись розділення на декілька паралельних гілок. Кожна з гілок може містити також розділення, послідовні дії або цикли.

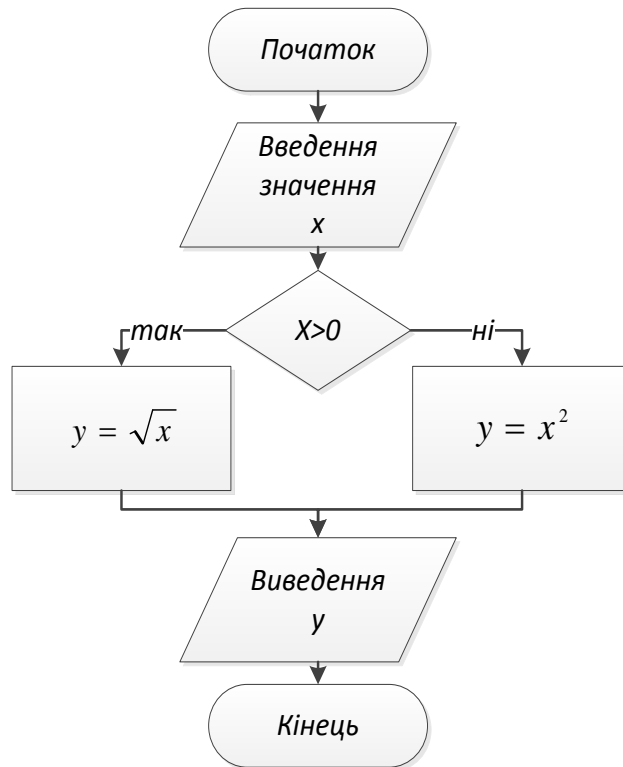


Рис.3.2. Приклад блок-схеми реалізації алгоритму з розгалуженням

- **Цикл** – інструкції що виконують одну і ту ж послідовність дій поки діє задана умова.

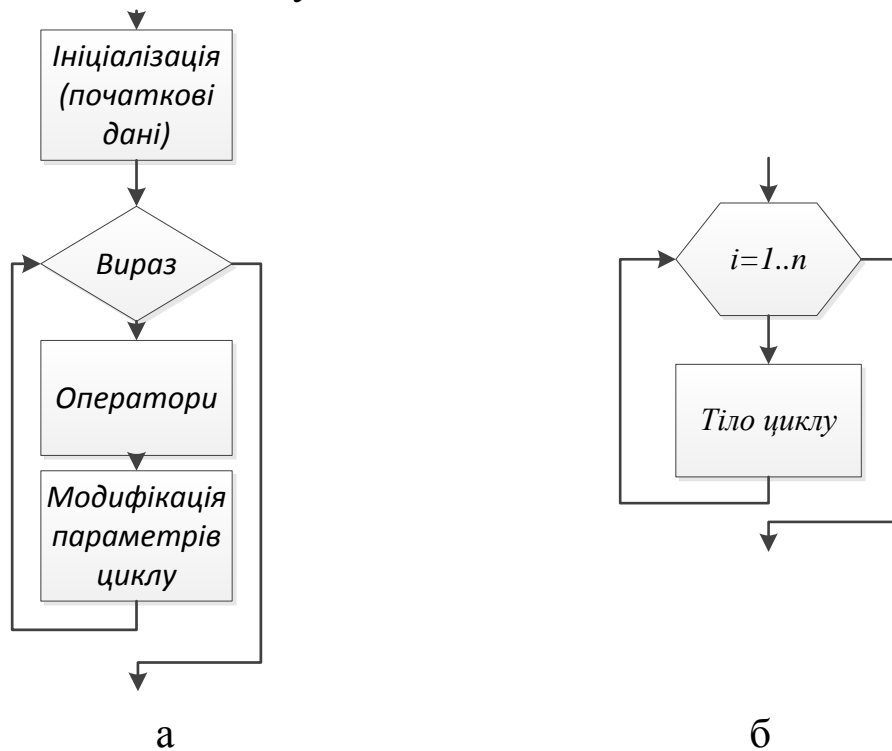


Рис.3.3. Приклад блок-схеми реалізації циклічного алгоритму (а – цикл з передумовою, б – цикл з лічильником)

### 3.2. Реалізація алгоритмів з розгалуженням

Хід виконання програми може бути лінійним, тобто таким, коли вирази виконуються, починаючи з першого і закінчуючи останнім, по порядку, не пропускаючи жодного рядка коду. Але частіше буває зовсім не так. При виконанні програмного коду деякі його ділянки можуть бути пропущені.

Припустимо, в реальному житті людина живе за розкладом (можна сказати, розклад – це своєрідний "програмний код", який слід виконати). У її розкладі о 18.00 стоїть похід в басейн. Однак людині надходить інформація, що басейн не працює. Цілком логічно скасувати своє заняття з плавання. Тобто однією з умов відвідування басейну повинно бути його функціонування, інакше повинні виконуватися інші дії.

Схожа нелінійність дій може бути і в комп'ютерній програмі. Частина коду повинна виконуватися лише при певному значенні конкретної умови. Найпростішою в Python для опису розгалужуючої структури, де дії виконуються лише у випадку істинності умови, є така конструкція:

#### **if ЛОГІЧНА\_УМОВА: ПОСЛІДОВНІСТЬ\_ВИРАЗІВ**

Цю конструкцію на блок-схемі можна зобразити:

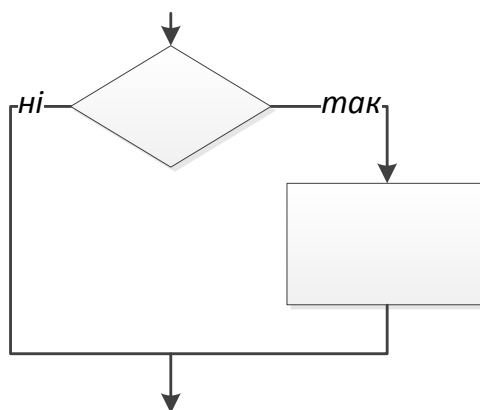


Рис.3.4. Блок-схема реалізації конструкції if

Першим йде ключове слово *if* (англ. "Якщо"); за ним – логічний вираз; потім двокрапка, що позначає кінець заголовка оператора, а після неї – будь-яка послідовність виразів або тіло

умовного оператора, яке буде виконуватися в разі, якщо умова в заголовку оператора істинна.

```
x = 2
if x > 0:
    print("x – додатне")
if x < 0:
    print("x – від’ємне")
```

Результатом запуску даного коду буде:

```
x – додатне
```

1. Привласнили значення 2 змінній *x*.
2. Зробили умовне порівняння за допомогою операторів *if*, виконуючи різні фрагменти коду в залежності від значень змінної *x*.

3. Викликали функцію *print()*, щоб вивести текст на екран.

Рядки *if* в Python є операторами, які перевіряють, чи є значення виразу (в даному випадку змінна *x*) рівним *True*.

*print()* – це вбудована в Python функція для виведення інформації. Вбудовані функції Python – це іменовані фрагменти коду, які виконують певні операції.

Кожен рядок *print()* відокремлений пробілами під відповідною перевіркою.

У більшості мов програмування символи начебто фігурних дужок (*{i}*) або ключові слова *begin* і *end* застосовується для того, щоб розбити код на розділи. У цих мовах хорошим тоном є використання відбиття пробілами, щоб зробити програму більш зрозумілою для себе та інших. Існують навіть інструменти, які допоможуть красиво вибудувати код.

Гвідо ван Росум при розробці Python вирішив, що виділення пробілами буде досить, щоб задати структуру програми і уникнути уведення дужок. Python відрізняється від інших мов тим, що пробіли в ньому використовуються для того, щоб задати структуру програми.

Як правило, використовують чотири пробіли для того, щоб виділити кожен підрозділ, хоча можна використовувати будь-яку кількість пробілів, Python чекає, що всередині одного розділу буде застосовуватися однакова кількість пробілів.

Рекомендований стиль – PEP-8 (<http://bit.ly/pep-8>) – використовувати чотири пробіли. Не рекомендується застосовувати табуляцію або поєднання табуляцій і пробілів – це заважає підраховувати відступи.

З огляду на це, в конструкції *if* код, який виконується при істинності умови, повинен обов'язково мати відступ вправо. Решта коду (основна програма) повинен мати той же відступ, що і слово *if*.

Зустрічається і більш складна форма розгалуження: *if-else*. Якщо умова при інструкції *if* є хибною, то виконується блок коду при інструкції *else*:

```
if ЛОГІЧНА_УМОВА:  
    ПОСЛІДОВНІСТЬ_ВИРАЗІВ_1  
else:  
    ПОСЛІДОВНІСТЬ_ВИРАЗІВ_2
```

Цю конструкцію на блок-схемі можна зобразити:

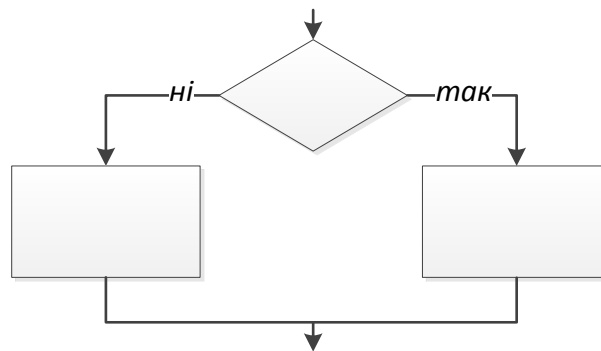


Рис.3.5. Блок-схема реалізації конструкції *if-else*

Працює ця конструкція наступним чином. Спочатку перевіряється перша умова і, якщо вона істинна, то виконується перша послідовність виразів. Якщо умова не виконується потік виконання переходить до рядка, який йде після *else*.

```

x = int(input("Введіть x="
"))
if x > 0:
    y=x**0.5
else:
    y=x**2
print("y =", y)

```

Отримаємо:

```

Введіть x= 9
y = 3.0

```

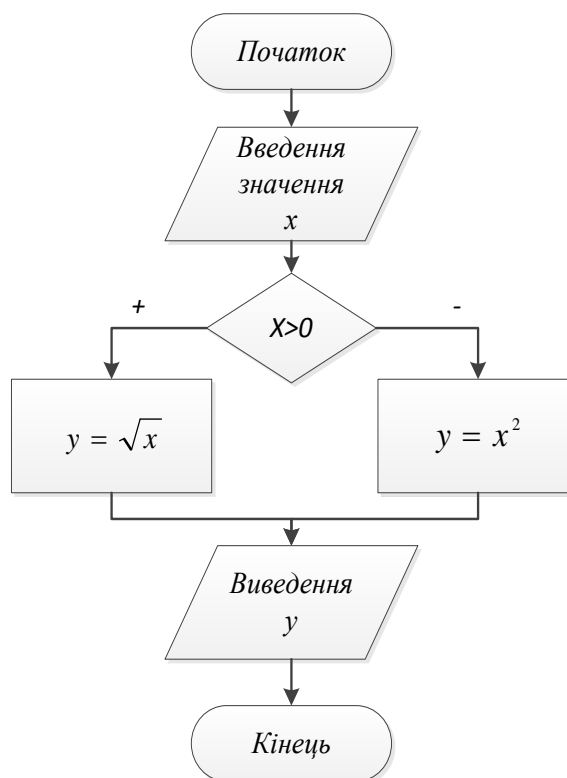


Рис.3.6. Приклад блок-схеми реалізації конструкції if-else

### *Альтернативні гілки програми*

Логіка програми що виконується може бути складнішою, ніж вибір однієї з двох гілок.

Умовний оператор *if* має розширений формат, що дозволяє перевіряти кілька незалежних одна від одної умов і виконувати один з блоків, поставлених у відповідність з цими умовами. У загальному вигляді оператор виглядає так:

```

if ЛОГІЧНА_УМОВА_1:
    ПОСЛІДОВНІСТЬ_ВИРАЗІВ_1
elif ЛОГІЧНА_УМОВА_2:
    ПОСЛІДОВНІСТЬ_ВИРАЗІВ_2
elif ЛОГІЧНА_УМОВА_3:
    ПОСЛІДОВНІСТЬ_ВИРАЗІВ_3
...
else:
    ПОСЛІДОВНІСТЬ_ВИРАЗІВ_N

```

Цю конструкцію на блок-схемі можна зобразити:

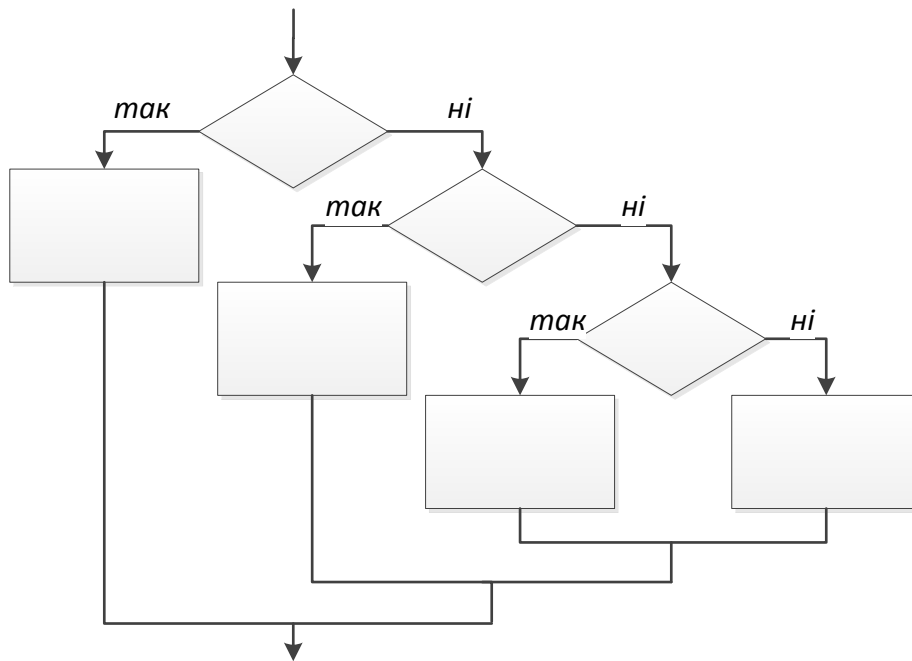


Рис.3.7. Блок-схема реалізації конструкції if-elif-else

Працює ця конструкція наступним чином. Спочатку перевіряється перша умова і, якщо вона істинна, то виконується перша послідовність виразів. Після цього потік виконання переходить до рядку, який йде після умовного оператора (тобто за послідовністю виразів  $N$ ). Якщо перша умова рівна *False*, то перевіряється друга умова (наступна після *elif*), і в разі його істинності виконується послідовність 2, а потім знову потік виконання переходить до рядка, наступного за оператором умови. Аналогічно перевіряються всі інші умови. До гілки програми *else* потік виконання доходить тільки в тому випадку, якщо не виконується жодна з умов.

Ключове слово *elif* походить від англ. "*Else if*" – "інакше якщо". Тобто умова, яка слідує після нього перевіряється тільки тоді, коли всі попередні умови хибні.

```
if not True:  
    print("1")  
elif not (1+1==3):  
    print("2")  
else:  
    print("3")
```



Результатом запуску даного коду буде:

3

### ***Оператор pass***

В процесі роботи над програмою слід намагатися після кожної зміни мати працюючу програму, але іноді не завжди відразу відомо, що необхідно виконати якщо умова приймає істинне значення, а що в протилежному випадку. В Python інструкції з розгалуженням, або цикли, або функції з порожнім тілом заборонені, тому в якості тіла використовується "порожній оператор" *pass*.

Припустимо, що заплановано використання умовного оператора з декількома умовами, але встигнуто написати тільки один з блоків умовного оператора. При цьому постає питання, як її налагодити, якщо програма не виконується через синтаксичну помилку.

```
if not True:
    print("1")
elif not (1+1==3):
elif not (1+1==4):
elif not (1+1==5):
```

Блоки для випадків, коли значення `not (1+1==3)`, `not (1+1==4)`, `not (1+1==5)`, ще не написані, тому програма не виконується через помилку `SyntaxError: expected an indented block`.

Ключове слово *pass* можна вставити на місце відсутнього блоку:

```
if not True:
    print("1")
elif not (1+1==3):
    pass
elif not (1+1==4):
    pass
elif not (1+1==5):
    pass
```

### 3.3. Реалізація циклічних алгоритмів

У реальному житті ми досить часто зустрічаємося з циклами. У комп'ютерних програмах поряд з інструкціями розгалуження (тобто вибором шляху дії) також існують інструкції циклів (повторення дії). Якби інструкцій циклу не існувало, довелось б багато разів вставляти в програму один і той же код поспіль стільки раз, скільки потрібно виконати однакоvu послідовність дій.

**Цикли** – це інструкції, які виконують одну й ту ж саму послідовність дій, поки діє задана умова.

Кожен циклічний оператор має тіло циклу – якийсь блок коду, який інтерпретатор буде повторювати поки умова повторення циклу буде залишатися істинною.

В програмуванні розрізняють такі види циклів:

- Цикл з передумовою – виконує дії поки умова є істинною (*while*).
- Цикл з післяумовою – спочатку виконуються команди, а потім перевіряється умова (*do...while*).
- Цикл з лічильником – виконується задану кількість разів (*for*).
- Сумісний цикл – виконує команди для кожного елемента із заданого набору значень (*for...which*).

В Python присутні лише цикл з передумовою (*while*) та цикл *for*, що поєднує в собі два види – цикл з лічильником та сумісний цикл.

#### **Оператор циклу *while***

Універсальним організатором циклу в мові програмування Python є конструкція *while* [5, 6, 7, 8, 9, 11]. Слово "while" з англійської мови перекладається як "поки" ("поки логічний вираз повертає істину, виконувати певні операції"). Конструкцію *while* на мові Python можна описати наступною схемою:

**while УМОВА\_ПОВТОРЕННЯ\_ЦИКЛУ:  
ТІЛО\_ЦИКЛУ**

Дуже схоже на оператор умови – тільки тут використовується інше ключове слово: *while* (англ. "Поки").

Дану конструкцію як правило використовують при повторі уведення даних користувачем, поки не буде отримано коректне значення:

```
correct_choice = False
while not correct_choice:
    choice = input("Введіть число 1 або 2:")
    if choice == "1" or choice == "2":
        correct_choice = True
    else:
        print ("Не правильно введено число, повторіть введення")
```

Результатом запуску даного коду буде:

```
Введіть число 1 або 2:3
Не правильно введено число, повторіть введення.
Введіть число 1 або 2:4
Не правильно введено число, повторіть введення
Введіть число 1 або 2:1
>>>
```

1. Визначили логічну змінну *correct\_choice*, присвоївши їй значення *False*.

2. Оператор циклу перевіряє умову *not correct\_choice*: заперечення *False* – істина. Тому починається виконання тіла циклу: виводиться запрошення "*Enter your choice, please (1 or 2):*" і очікується уведення користувача.

3. Після натискання клавіші *Enter* введене значення порівнюється з рядками "*1*" і "*2*", і якщо воно дорівнює одній з цих значень, то змінній *correct\_choice* присвоюється значення *True*. В іншому випадку програма виводить повідомлення "*Не правильно введено число, повторіть введення*".

4. Оператор циклу знову перевіряє умову і якщо вона як і раніше істинна, то тіло циклу повторюється знову, інакше потік виконання переходить до наступного оператору, і інтерпретатор виходить з циклу і виконання програми припиняється або виконуються дії що прописані поза тілом циклу.

Ще один варіант використання оператора циклу – обчислення формул із змінним параметром.

$$\sum_{i=1}^n i^3 = 1^3 + \dots + n^3$$

В даному випадку, параметром, що змінюється є  $i$ , причому  $i$  послідовно приймає значення в діапазоні від  $1$  до  $n$ .

За допомогою оператора циклу *while* рішення буде виглядати так:

```
n = int(input("Введіть n: "))
sum = 0
i = 1
while i <= n:
    sum += i**3
    i += 1
print ("sum = ", sum)
```

Результатом запуску даного коду буде:

```
Введіть n: 5
sum = 225
```

1. Необхідно ввести  $n$  – граничне значення  $i$ .
2. Ініціалізація змінної  $sum$  – в ній буде зберігатися результат, початкове значення –  $0$ .
3. Ініціалізація змінної  $i$  (лічильника  $i$ ) – за умовою, початкове значення –  $1$ .
4. Починається цикл, який виконується, поки  $i \leq n$ .
5. У тілі циклу в змінну  $sum$  записується сума значення з цієї змінної, отриманої на попередньому кроці, і значення  $i$ , піднесеної в куб.
6. Лічильнику  $i$  присвоюється наступне значення.
7. Після завершення циклу виводиться значення  $sum$  після виконання останнього кроку.

Наступне значення лічильника отримують додаванням до його поточного значення кроку циклу (в даному випадку крок циклу дорівнює  $1$ ). Крок циклу при необхідності може бути від’ємним, і навіть дробовим. Крім того, крок циклу може

змінюватися на кожній ітерації (тобто при кожному повторенні тіла циклу).

### ***Нескінченні цикли***

Іноді можна зіткнулися з проблемою нескінченного повторення блоку коду, що виникає, наприклад, через семантичну помилку в програмі:

```
i = 0
while i < 10:
    print (i)
```

Такий цикл буде виконуватися нескінченно, тому що умова  $i < 10$  завжди буде істинною, адже значення змінної  $i$  не змінюється: така програма буде виводити нескінченну послідовність нулів.

У циклів немає обмеження кількості повторень тіла циклу, тому програма з нескінченим циклом буде працювати безперервно, поки не буде зроблено аварійної зупинки натисканням комбінації клавіш Ctrl+C, не зупинимо відповідний процес засобами операційної системи, або не будуть вичерпані доступні ресурси комп'ютера (наприклад, може бути досягнуто максимально допустимої кількості відкритих файлів), чи не виникне виняток. У таких ситуаціях кажуть, що програма зациклилася.

Знайти зациклення в програмі іноді буває не так-то просто, адже в реальних програмах зазвичай використовується багато циклів, кожен з яких потенційно може виконуватися нескінченно.

Проте, відсутність обмеження на кількість повторів тіла циклу дає нові можливості. Багато програм представляють собою цикл, в тілі якого проводиться відстеження та оброблення дій користувачів або запитів з інших програмних і автоматизованих систем. При цьому такі програми можуть працювати без перебоїв дуже тривалі терміни, іноді роками.

Цикли – це дуже потужний засіб реалізації, але вони вимагають деякої уважності.

Якщо необхідно, щоб цикл виконувався до тих пір, поки щось не станеться, але точно не відомо, коли ця подія трапиться,

можна скористатися нескінченним циклом, що містить оператор *break*.

```
i = 1

while True:
    if i > 5:
        break           # Перериваємо цикл
    print (i)
    i += 1
```

Результатом запуску даного коду буде:

```
1
2
3
4
5
```

1. Виведемо 5 чисел починаючи з 1. Змінній *i* привласнено початкове значення 1.

2. Найпростішою умовою для нескінченного циклу є значення True.

3. Виконується перевірка значення змінної *i* зі значенням 5. Якщо воно співпадає або є більшим, то виконується оператор *break* що перериває виконання циклу.

4. Виводиться на екран значення числа (виконується якщо не спрацював оператор *break*).

5. Збільшується значення змінної *i* на 1

Іноді потрібно не переривати весь цикл, а лише пропустити по якійсь причині одну ітерацію. *Continue* дозволяє перейти до наступної ітерації циклу до завершуючи виконання всіх виразів всередині циклу.

Розглянемо приклад: виводяться на екран цілі числа в діапазоні від 1 до 10, крім чисел 3, 4, 5.

```
i=0

while i<=10:
    i+=1
```

```
if 3<=i<=5:
    continue
print(i)
```

Результатом запуску даного коду буде:

```
1
2
6
7
8
9
10
11
```

### *Альтернативна гілка else*

Мова Python дозволяє використовувати розширений варіант оператора циклу:

```
while УМОВА_ПОВТОРЕННЯ_ЦИКЛУ:
    ТІЛО_ЦИКЛУ
else:
    АЛЬТЕРНАТИВНА_ГІЛКА_ЦИКЛУ
```

Поки виконується умова повторення тіла циклу, оператор *while* працює так само, як і в звичайному варіанті, але як тільки умова повторення перестає виконуватися, потік виконання направляється по альтернативній гілці *else* – так само, як в умовному операторі *if*, вона виконається всього один раз.

```
i = 0
while i < 3:
    print (i)
    i += 1
else:
    print ("кінець циклу")
```

Результатом запуску даного коду буде:

```
0
1
```

2

кінець циклу

### *Цикл for*

В Python ітератори часто використовуються оскільки вони дозволяють проходити структури даних, не знаючи, наскільки ці структури великі і як реалізовані. Можливо пройти по послідовності таким чином:

```
numbers = [1, 2, 3, 4, 5]
i = 0

while i <=len(numbers)-1:
    print(numbers[i])
    i+=1
```

Результатом запуску даного коду буде:

```
1
2
3
4
5
```

Однак існує більш характерний для Python спосіб вирішення цього завдання:

```
for element in numbers:
    print(element)
```

Результатом запуску даного коду буде:

```
1
2
3
4
5
```

Списки подібні до *numbers* є одними з ітераційних об'єктів в Python поряд з рядками, кортежами, словниками та деякими іншими елементами.



Ітераційні об'єкти – це ті, вміст яких можна перебрати в циклі. Ітерування по кортежу або списку повертає один елемент за раз. Ітерування по рядку повертає один символ за раз:

```
word = 'cat'
for letter in word:
    print(letter)
```

Результатом запуску даного коду буде:

```
c
a
t
```

Цикл *for* дозволяє перебирати всі елементи вказаної послідовності (список, кортеж, рядок). Цикл спрацює рівно стільки разів, скільки елементів знаходиться в послідовності.

### **for ЗМІННА in ПОСЛІДОВНІСТЬ: ТІЛО\_ЦИКЛУ**

Ім'я змінної в яку на кожному кроці буде розміщено елемент послідовності обирається довільно.

Оператори *break* та *continue* працюють так само як і в циклі *while*. Блок *else* дозволяє перевірити чи виконався цикл *for* повністю якщо ключове слово *break* не було викликане, то буде виконаний блок *else*. Це корисно, якщо потрібно переконатися в тому, що попередній цикл виконався повністю, замість того щоб рано перерватися:

```
numbers = []
for number in numbers:
    print('Цей список має деякий елемент', number)
    break
else:
    # Відсутність переривання означає, що елемент відсутній

    print('Елементів немає в списку, чи не так?')
```

Результатом запуску даного коду буде:

```
'Елементів немає в списку, чи не так?'
```

В циклах використання блоку *else* може здатися нелогічним. Можна розглядати цикл як пошук чогось, в такому випадку *else* буде викликатися, якщо нічого не було знайдено.

### ***Ітерування за кількома послідовностям за допомогою функції zip()***

Щоб паралельно ітеруватися за кількома послідовностям одночасно можна використати функцію *zip()*:

```
days = ['Monday', 'Tuesday', 'Wednesday']
fruits = ['banana', 'orange']
drinks = ['coffee', 'tea', 'beer']
for day, fruit, drink in zip(days, fruits, drinks):
    print(day, ": drink", drink, "eat", fruit)
```

Результатом запуску даного коду буде:

```
Monday : drink coffee eat banana
Tuesday : drink tea eat orange
```

Функція *zip()* припиняє свою роботу, коли виконується найкоротша послідовність. Один зі списків (*fruits*) виявився коротшим за інші, тому результат було виведено лише для двох елементів.

Функція *zip()* – ітератор, який повертає кортежі, що складаються з відповідних елементів аргументів-послідовностей.

### ***Генерація числових послідовностей за допомогою функції range()***

Функція *range()* повертає послідовність чисел в заданому діапазоні без необхідності створювати і зберігати велику структуру даних на зразок списку або кортежу.

Це дозволяє створювати великі діапазони, не використавши всю пам'ять комп'ютера і не обірвавши виконання програми.

***range (start, end, step)***

Якщо опустити значення *start*, діапазон почнеться з 0. Необхідною є лише значення *end*, що визначає останнє значення, яке буде створено прямо перед зупинкою функції. Значення

step по замовчуванню дорівнює 1, але можна змінити його на -1 [5 - 13].

Якщо просто викликати функцію *range()*, то результату не буде отримано.

```
>>> range(0,10,1)
range(0, 10)
```

Як і *zip ()*, функція *range ()* повертає ітераційний об'єкт, тому потрібно пройти за значеннями за допомогою конструкції *for ... in* або перетворити об'єкт в послідовність (список, кортеж та ін.).

```
for x in range(0, 3):
    print(x)
```

Результатом запуску даного коду буде:

```
0
1
2
```

```
for x in range(2, -1, -1):
    print(x)
```

Результатом запуску даного коду буде:

```
2
1
0
```

### ***Підходи до створення списків***

Створити список цілих чисел можна декількома способами.

Можна додавати елементи до списку по одному за раз використовуючи функцію *append()*:

```
number_list = []
print (number_list)

number_list.append(1)
print (number_list)

number_list.append(2)
```

```
number_list.append(3)
number_list.append(4)
number_list.append(5)
print (number_list)
```

Результатом запуску даного коду буде:

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

Можна використати ітератор та функцію *range()*:

```
number_list = []
for number in range(1, 6):
    number_list.append(number)
print (number_list)
```

Отримаємо:

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

Або ж перетворити в список сам результат роботи функції *range()*:

```
number_list = list(range(1, 6))
print (number_list)
```

Отримаємо:

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

Однак можна створити список за допомогою включення списку.

### ***Спискове включення***

***Включення*** – це компактний спосіб створити структуру даних з одного або більше ітераторів. Включення дозволяють вам об'єднувати цикли і умовні перевірки, не використовуючи при цьому громіздкий синтаксис.

Найпростіша форма такого включення виглядає так:

**[ВИРАЗ for ЕЛЕМЕНТ in ІТЕРАЦІЙНИЙ\_ОБ'ЄКТ]**

```
number_list = [number for number in range(1,6)]
print (number_list)
```

Отримаємо:

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

У першому рядку потрібно, щоб перша змінна *number* сформувала значення для списку: слід розмістити результат роботи циклу в змінну *number\_list*. Друга змінна *number* є частиною циклу *for*. Щоб показати, що перша змінна *number* є виразом, спробуємо такий варіант:

```
number_list = [number-1 for number in range(1,6)]
print (number_list)
```

Отримаємо:

```
[0, 1, 2, 3, 4]
```

Включення списку може містити умовний вираз:

## **[ВИРАЗ for ЕЛЕМЕНТ in ІТЕРАЦІЙНИЙ\_ОБ'ЄКТ if УМОВА]**

Наступне включення створює список, що складається тільки з парних чисел, розташованих в діапазоні від 1 до 5 (вираз *number% 2* має значення *True* для парних чисел і *False* для непарних):

```
number_list = [number for number in range(1,6) if number % 2
== 1]
print (number_list)
```

Отримаємо:

```
[1, 3, 5]
```

Вираз може містити будь-що. Для створення списку з елементів, що будуть вводиться з клавіатури:

```
number_list = [int(input('введіть число: ')) for number in
range(int(input('введіть кількість елементів: ')))]
print (number_list)
```

Отримаємо:

```
введіть кількість елементів: 5
введіть число: 1
введіть число: 0
введіть число: 5
введіть число: 1
введіть число: 3
[1, 0, 5, 1, 3]
```

### ***Вкладені списки***

Списки можуть містити елементи різних типів, включаючи інші списки. ***Вкладеними*** називаються списки, які є елементами іншого списку.

Вкладені списки зазвичай використовуються для подання матриць. Змінною `list_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]` описано матрицю:

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

В змінній `list_list` зберігається список з трьома елементами, в кожному з яких зберігається рядок матриці теж у вигляді списків. Витягти рядок можна за допомогою оператора індексування:

```
list_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(list_list)

print(list_list [0])
```

Отримаємо:

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

Щоб звернутися або отримати елемент матриці (вкладеного списку) необхідно вказати два індекси:

```
list_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

print(list_list)
```

```
print(list_list [0])
print(list_list [0][1])
```

Отримаємо:

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

Оператори індексування виконуються зліва на право. Індекс [0] посилається на перший елемент списку `list_list`, а [1] – на другий елемент внутрішнього списку. Перший індекс визначає номер рядка, а другий – номер елемента (тобто номер стовпчика) в матриці.

### ***Вкладені цикли***

Цикли можуть бути вкладені один в одного. При цьому цикли можуть використовувати різні змінні-лічильники.

Найпростіше застосування вкладених операторів циклу – побудова двовимірних таблиць (матриць, масивів), наприклад:

```
i = 1

while i <= 10:
    j = 1
    while j <= 10:
        print (i * j, end="\t")
        j += 1
    print ()
    i += 1
```

Цикл, перевіряючий умову  $i \leq 10$ , відповідає за повторення рядків таблиці. У його тілі виконується другий цикл, який виводить добуток  $i$  і  $j$  10 разів поспіль, розділяючи знаками табуляції отримані результати (`end="\t"`).

Функція `print()` без параметра виконує перехід на новий рядок.

В результат виконання даної програми отримаємо

наступну таблицю:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Перший стовпець і перший рядок містять числа від 1 до 10. На перетині  $i$ -того рядка і  $j$ -того стовпця знаходиться добуток  $i$  і  $j$ . Отримуємо таблицю множення.

Якщо додати ще один цикл, то отримаємо тривимірну таблицю добутоків трьох чисел. Таким чином вкладені цикли дозволяють будувати таблиці будь-якої розмірності, що дуже часто застосовується в складних наукових розрахунках.

```
outer = [1, 2, 3]           # Зовнішній цикл
inner = [4, 5, 6]          # Вкладений (внутрішній) цикл
for i in outer:
    for j in inner:
        print ('i=', i, 'j=', j)
```

Результатом запуску даного коду буде:

```
i= 1 j= 4
i= 1 j= 5
i= 1 j= 6
i= 2 j= 4
i= 2 j= 5
i= 2 j= 6
i= 3 j= 4
i= 3 j= 5
i= 3 j= 6
```

Цикл *for* спочатку просувається по елементах зовнішнього циклу (фіксуємо  $i=1$ ), потім переходить до вкладеного циклу



(змінна  $j$ ) і проходимо по всіх елементах вкладеного списку. Далі повертаємося до зовнішнього циклу (фіксуємо значення  $i=2$ ) і знову проходимо по всіх елементах вкладеного списку. Так повторюємо до тих пір, поки не закінчатся елементи в зовнішньому списку:

Даний прийом активно використовується при роботі з вкладеними списками.

Спочатку приклад з одним циклом *for*:

```
lst = [[1, 2, 3], [4, 5, 6]]
for i in lst:
    print (i)
```

Результатом запуску даного коду буде:

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

У прикладі за допомогою циклу *for* перебираються всі елементи списку, які також є списками.

Якщо необхідно дістатися до елементів вкладених списків, то доведеться використовувати вкладений цикл *for*:

```
lst = [[1, 2, 3], [4, 5, 6]]
for i in lst:          # Цикл за елементами зовнішнього
    сплиску
    print()
    for j in i:        # Цикл по елементах елементів
    зовнішнього списку
        print (j, end="")
```

Результатом запуску даного коду буде:

```
123
456
```

### **Завдання на комп'ютерний практикум**

1. Проаналізувати варіант завдання. Скласти програму у відповідності до варіанту та побудувати блок-схему.

Напишіть програму, яка обчислює значення функції (на вхід подається дійсне число):

$Y = \begin{cases} 2x^2 + a & x \leq 0; a=3,5bx; b=-0,9 \\ (x+3a)c & x > 0; c=1,35x+b\sqrt{x} \end{cases}$
--

2. Обчислити значення функції на відрізку  $[x_0; x_k]$  з кроком  $dx$  за допомогою циклу з `while`.
3. Знайти суму  $n$  членів ряду, заданого за варіантом.
 
$$\sum_{i=1}^n \frac{\sqrt{2i}}{\sqrt{i+5} - \sqrt{i}}$$
4. Скласти програму яка за введеними з клавіатури значеннями змінних, виконає необхідні розрахунки.
  - Знайти суму цифр заданого натурального числа  $n$ .
  - Дано натуральне число. Підрахувати загальну кількість його дільників.

### Запитання для самоконтролю

1. Як описується та виконується оператор розгалуження?
2. Як описується та виконується оператор множинного розгалуження?
3. Що називається логічним виразом?
4. Які 3 можливих варіанти представлення умови в інструкції `if`?
5. Що таке цикл? Навіщо вони потрібні?
6. Як описується та виконується циклічна інструкція `while`?
7. Як можна організувати нескінченні цикли? Наведіть декілька варіантів і поясніть їх.
8. Як можна вийти з нескінченних циклів?
9. Що відбувається при запуску нескінченного циклу?
10. Чи може оператор циклу не мати тіла? Чому?
11. Для чого служать оператори переривання `break` та `continue`? Наведіть приклад.
12. Як працює оператор `for`?
13. Для організації яких циклів застосовується оператор `for`?
14. Що таке масиви? Як розташовуються елементи масивів у пам'яті?
15. Як звернутись до першого та останнього елементу масиву?

## РОЗДІЛ 4. ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ

### 4.1. Функції

Зазвичай реалізація складних задач містить величезні фрагменти коду. Зручним способом організувати великий фрагмент коду в більш зручні фрагменти є створення функцій. Функції в Python є основою при написанні програм.

Іноді функцію порівнюють з "чорним ящиком", коли відомо, що на вході і що при цьому на виході, а нутроці "чорного ящика" часто бувають приховані.

Існує велика кількість вбудованих функцій. Наприклад функція *abs()*, приймає на вхід один аргумент – об'єкт числового типу і повертає абсолютне значення для цього об'єкта.

```
>>> abs (-9)
9
```

Результат виклику функції можна присвоїти змінній, використовувати його в якості операндів математичних виразів, тобто складати більш складні вирази.

```
x = abs (-15)
y = x +5

print('y=', y)
```

Результатом запуску даного коду буде:

```
20
```

Крім складання складних математичних виразів Python дозволяє передавати результати виконання функцій в якості аргументів інших функцій без використання додаткових змінних:

```
print(abs (-15))
```

Отримаємо:

```
15
```

Спочатку визначається абсолютне значення цілого числа -15, а потім за допомогою функції *print()* виводиться на екран результат розрахунку.

Можна реалізовувати власні функції.

**Функція** – це іменований фрагмент коду, відокремлений від інших. Вона може приймає будь-яку кількість будь-яких вхідних параметрів і повертати будь-яку кількість будь-яких результатів.

З функцією можна зробити дві речі:

- визначити;
- викликати.

Щоб визначити функцію, використовується наступна конструкція:

### **def ІМ'Я\_ФУНКЦІЇ (ВХІДНІ\_ПАРАМЕТРИ): ФУНКЦІЯ**

Імена функцій підкоряються тим же правилам, що і імена змінних (вони повинні починатися з літери або `_` і містити тільки букви, цифри або `_`).

Функція може не містити параметри але круглі дужки все рівно необхідно вказувати:

```
def do_nothing():  
... pass
```

Тіло функції відділяється пробілами. Використання виразу *pass* відображає, що функція нічого не робить.

Щоб викликати функцію вказується її ім'я та дужки з параметрами:

```
do_nothing()
```

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

Визначимо і викличемо функцію, яка не має параметрів і виводить на екран одне слово:

```
def salute():  
    print('Hi!')  
salute()
```

Отримаємо:

```
Hi!
```

Коли викликається функція `salute()`, Python виконує код, розташований всередині її опису. У цьому випадку він виводить одне слово і повертає управління основній програмі.

### **Параметри функції**

Функція може приймати параметри та повертати значення. Параметри функції – звичайні змінні, якими функція користується для внутрішніх розрахунків. Якщо параметрів декілька – вони перераховуються через кому [5 – 10, 13].

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

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

```
def print_numbers(limit):
    for i in range (limit):
        print(i)

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

print_numbers(n)
```

Результатом запуску даного коду буде:

```
Введіть кількість елементів: 5
0
1
2
3
4
```

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

Існують функції які просто щось виконують, наприклад вбудована функція `print()` яка виводить на екран певні значення:

```
n=15  
print(n)
```

Отримаємо:

```
15
```

А є функції які повертають розраховане значення, що може бути привласнене змінній, наприклад вбудована функція `input()`:

```
a=input('Введіть слово: ')
```

Отримаємо:

```
Введіть слово: Ні!
```

Для того щоб переглянути результат роботи такої функції потрібно скористатися функцією `print()`

```
a=input('Введіть слово: ')  
print(a)
```

Отримаємо:

```
Ні!
```

При необхідності повернути результат роботи функції в програму, з якої вона викликалася, для її подальшого оброблення застосовується команда ***return***. Вираз, що стоїть після *return* буде повертатися в якості результату виклику функції.

Без аргументів *return* використовується для виходу з функції (інакше вихід відбудеться при досягненні кінця функції).

В Python функції здатні повертати кілька значень одночасно.

```
def PrintRoots(a, b, c):  
    D = b**2 - 4 * a * c  
    import math  
    x1 = (-b + math.sqrt(D)) / 2 * a  
    x2 = (-b - math.sqrt(D)) / 2 * a  
    return x1, x2
```

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

Результатом запуску даного коду буде:

```
(1.0, -1.0)
```

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

```
x1, x2 = PrintRoots(1.0, 0, -1.0)
print("x1 =", x1, "\nx2 =", x2)
```

Результатом запуску даного коду буде:

```
x1 = 1.0
x2 = -1.0
```

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

```
def traffic_light (color):
    if color == 'red':
        return "STOP!"
    elif color == "green":
        return "GO!"
    elif color == 'yellow':
        return "GET READY!"
    else:
        return "Broken traffic light!"
```

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

```
result = traffic_light('blue')
print(result)
```

Функція зробить наступне:

- присвоїть значення 'blue' параметру функції color;
- пройде по логічному ланцюжку if-elif-else;
- поверне рядок;
- присвоїть рядок змінній result.

Результатом буде:

```
'Broken traffic light!'
```

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

```
def do_nothing():  
    pass
```

Результатом буде:

```
None
```

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

### ***Простори імен та області видимості***

Кожна функція визначає власний простір імен. Якщо визначити змінну, яка називається *x* в основній програмі та іншу змінну *x* в окремій функції, то вони будуть посилатися на різні значення. В основній програмі визначається глобальний простір імен, а змінні що тут знаходяться називаються ***глобальними*** тобто до неї можна звернутися з будь якого місця програми, в тому числі і всередині функції. Змінна є ***локальною*** (видно тільки всередині функції), якщо значення їй присвоюється всередині функції [5, 8, 9].

```
a = 3 # Глобальна змінна  
y = 8 # Глобальна змінна  
  
def func ():  
    print ('func: глобальна змінна a = ', a)  
    y = 5 # Локальна змінна  
    print ('func: локальна змінна y = ', y)  
  
func () # Виклик функції func()  
  
print ('??? y = ', y) # Відобразиться глобальна  
змінна
```



```
print ('глобальна змінна a = ', a)
print ('глобальна змінна y = ', y)
```

Результатом запуску даного коду буде:

```
func: глобальна змінна a = 3
func: локальна змінна y = 5
??? y = 8
глобальна змінна a = 3
глобальна змінна y = 8
```

Всередині функції можна звернутися до глобальної змінної *a* і вивести її значення на екран. Далі всередині функції створюється локальна змінна *y*, причому її ім'я збігається з ім'ям глобальної змінної – в цьому випадку при зверненні до *y* виводиться вміст локальної змінної, а глобальна залишається незмінною.

Щоб змінити значення глобальної змінної всередині функції використовується ключове слово *global*.

```
x = 50                                # Глобальна змінна

def func():
    global x                            # Вказуємо, що x-глобальна
    змінна
    print('x =', x)
    x = 2                                # Змінюємо глобальну змінну
    print('Замінюємо глобальне значення x на', x)
func()

x = 50
print('Значення x =', x)
```

Результатом запуску даного коду буде:

```
x = 50
Замінюємо глобальне значення x на 2
Значення x = 50
```

З академічної точки зору зміна глобальної змінної всередині функції порушує принципи модульності програми.

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

```
def summa (x, y):  
    return x + y  
  
print(summa(5,6))  
  
f = summa  
v = f (10, 3)          # Викликаємо функцію з іншим ім'ям  
print(v)
```

Результатом запуску даного коду буде:

```
11  
13
```

### *Аргументи функцій*

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

### *Позиційні аргументи*

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

```
def func(a, b, c):  
    return a+b*c  
  
print(func(1, 2, 3))          # a = 1, b = 2, c = 3
```

Отримаємо:

```
7
```

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

## ***Іменовані аргументи***

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

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

Отримаємо:

```
5
```

Можна об'єднувати позиційні аргументи та іменовані аргументи.

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

Отримаємо:

```
8
```

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

## ***Значення параметра за замовчуванням***

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

```
def func(a, b, c=2):  
    return a+b*c
```

Викликаючи функцію *func()* можна не передавати їй аргумент *c*:

```
print(func(1, 2))
```

Отримаємо:

```
5
```

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

```
print(func(1, 2, 3))
```

Отримаємо:

```
7
```

### **Отримання позиційних аргументів**

Якщо перед параметром у визначенні функції вказати символ \*, то функції можна буде передати будь-яку кількість параметрів. Всі передані параметри зберігаються в кортежі.

У наступному прикладі `args` є кортежем параметрів, який був створений з аргументів, переданих у функцію `print_args()`:

```
def print_args(*args):  
    # функція приймає будь-яку кількість параметрів  
  
    print('Кортеж позиційних аргументів:', args)
```

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

```
print_args()  
Кортеж позиційних аргументів: ()
```

Всі аргументи, які будуть передані, виведуться на екран як кортеж `args`:

```
print_args(1, 2, 3, 'Hi!')  
Кортеж позиційних аргументів: (1, 2, 3, 'Hi!')
```

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

```
def print_more(num_1, num_2, *args):  
    print(num_1)  
    print(num_2)  
    print(args)  
  
print_more(1, 2, 3, 4, 5, 6)
```

Результатом запуску даного коду буде:

```
1
2
(3, 4, 5, 6)
```

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

### **Отримання іменованих аргументів**

Можна використовувати `**`, щоб згрупувати іменовані аргументи в словник, де імена аргументів стануть ключами, а їх значення – відповідними значеннями в словнику. У наступному прикладі визначається функція `print_kwargs` (), в якій виводяться її іменовані аргументи:

```
def print_kwargs(**kwargs):
    print('Іменовані аргументи:', kwargs)
```

Викликавши її, передавши кілька аргументів:

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

Отримаємо наступний результат:

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

У середині функції `kwargs` є словником.

Якщо використано позиційні аргументи та іменовані аргументи (`*args` і `**kwargs`), вони повинні слідувати в цьому ж порядку. Як і у випадку з `args`, не обов'язково називати цей словник `kwargs`.

### **Документаційні рядки**

Можна додавати документацію до власних функцій, модулів, класів, заключивши рядок на початку тіла функції у лапки. Вона називається **рядком документації** або **документаційним рядком** (`docstring`):

```
def func(anything):
    'Функція повертає введений аргумент'
    return anything
```

Як правило документація містить розгорнуту інформацію про те, що дана функція (модуль) виконує, які аргументи приймає та що повертає в результаті виконання, опис всіх констант (функцій, для модуля). Тож документація може бути досить великого розміру і щоб використати до такої інформації форматування та вивести багато рядків коментарів необхідно заключити документаційний рядок в три пари подвійних лапок.

```
def print_if_true(thing, check):  
    """  
    Prints the first argument if a second argument is true.  
    The operation is:  
        1. Check whether the *second* argument is true.  
        2. If it is, print the *first* argument.  
    """  
    if check:  
        print(thing)  
print(help(print_if_true))
```

На відміну від звичайних коментарів, до документаційних рядків можна звернутися під час виконання програми. Для того щоб вивести рядок документації деякої функції, необхідно викликати функцію *help()*, передати їй ім'я функції, щоб отримати список всіх аргументів і відформатований рядок документації:

```
Help on function print_if_true in module __main__:  
  
print_if_true(thing, check)  
    Prints the first argument if a second argument is true.  
    The operation is:  
        1. Check whether the *second* argument is true.  
        2. If it is, print the *first* argument.  
  
None
```

Щоб відобразити рядок документації без форматування:

```
def print_if_true(thing, check):  
    """
```

Prints the first argument if a second argument is true.

The operation is:

1. Check whether the *\*second\** argument is true.
2. If it is, print the *\*first\** argument.

'''

if check:

print(thing)

print(func.\_\_doc\_\_)

Отримаємо:

Prints the first argument if a second argument is true.

The operation is:

1. Check whether the *\*second\** argument is true.
2. If it is, print the *\*first\** argument.

Рядок `__doc__` є внутрішнім ім'ям рядка документації як змінної всередині функції.

### ***Потік виконання***

З появою функцій програми перестали бути лінійними, в зв'язку з цим виникло поняття ***потіку виконання*** – послідовності виконання інструкцій, що складають програму.

Виконання програми, написаної на Python завжди починається з першого виразу, а наступні вирази виконуються один за іншим зверху вниз. Причому, визначення функцій ніяк не впливають на потік виконання, тому що тіло будь-якої функції не виконується до тих пір, поки не буде викликана відповідна функція.

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

З тіла будь-якої функції може бути викликана інша функція, яка теж може в своєму тілі містити виклики функцій і т.д. Проте, інтерпретатор Python пам'ятає звідки була викликана кожна функція, і рано чи пізно, якщо під час виконання не виникне

ніяких винятків, він повернеться до вихідного виклику, щоб перейти до наступної інструкції.

```
def func1(name):  
    print('Привіт, '+name)  
  
def func2():  
    return input('Введіть ім\'я ')  
  
func1(func2())
```

Результатом запуску даного коду буде:

```
Введіть ім'я Alex  
Привіт, Alex
```

### ***Внутрішні функції***

Можна визначити функцію всередині іншої функції:

```
def outer(a, b):  
    def inner(c, d):  
        return c + d  
    return inner(a, b)  
  
print(outer(4, 7))
```

Результатом запуску даного коду буде:

```
11
```

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

### ***Анонімні функції: функція lambda()***

В Python лямбда-функція – це анонімна функція, виражена одним виразом. Її можна використовувати замість звичайної маленької функції.

Для того щоб проілюструвати анонімні функції, спочатку створимо приклад, в якому використовуються звичайні функції. Визначимо функцію *edit\_story()*. Вона має такі аргументи:



- words – список слів;
- func – функція, яка повинна бути застосована до кожного слова в списку words.
- stairs – список слів
- edit\_story – функція, яку потрібно застосувати кожного слова списку (записує з великої літери кожне слово і додає знак оклику):

```
def edit_story(words, func):
    for word in words:
        print(func(word))

def enliven(word):
    return word.capitalize() + '!'

stairs = ['hi', 'hello', 'привіт']

(edit_story(stairs, enliven))
```

Результатом запуску даного коду буде:

```
Hi!
Hello!
Привіт!
```

Функцію *enliven()* можна замінити лямбда функцією:

```
def edit_story(words, func):
    for word in words:
        print(func(word))

stairs = ['hi', 'hello', 'привіт']

edit_story(stairs, lambda word: word.capitalize() + '!')
```

Результатом запуску даного коду буде:

```
Hi!
Hello!
Привіт!
```

Лямбда приймає один аргумент, який в цьому прикладі названий `word`. Все, що знаходиться між двокрапкою та закриваючою дужкою, є визначенням функції.

Часто використання справжніх функцій на зразок `enliven()` набагато прозоріше, ніж використання лямбда. Лямбда найбільш корисні у випадках, коли потрібно визначити багато дрібних функцій і запам'ятати усі їх імена.

## 4.2. Рекурсія

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

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

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

$$n! = \begin{cases} 1 & n = 0 \\ n * (n - 1)! & n > 0 \end{cases}$$

```
def factorial (n):  
    if n>0:  
        return n* factorial(n-1)  
    else:  
        return 1
```

```
print(factorial (5))
```

Отримаємо:

```
120
```

$$x^n = \begin{cases} 1 & n = 0 \\ x * x^{n-1} & n > 0 \end{cases}$$

```
x=10  
def rec_func (n):  
    if n>0:
```

```
    return x* rec_func (n-1)
else:
    return 1

print(rec_func (5))
```

Отримаємо:

```
100000
```

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

$$S = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} = \sum_{i=1}^n \frac{1}{i}$$

```
def rec_func_2(n, i):
    if i==n:
        return 1/n
    else:
        return 1/i+rec_func_2(n, i+1)

print(rec_func_2(5, 1))
```

Отримаємо:

```
2.2833333333333333
```

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

```
rec_func_2(5, 1);
```

При виконанні тіла функції сформується наступне:

$$\frac{1}{1} + \text{rec\_func\_2}(5,2)$$

Що знову змушує звернутися до функції  $\text{rec\_func\_2}(5, 2)$ , що призводить до появи нового значення:

$$\frac{1}{2} + \text{rec\_func\_2}(5,3)$$

Після виконання ще двох звернень ситуація виявиться наступною:

$$\frac{1}{3} + \text{rec\_func\_2}(5, 4)$$

$$\frac{1}{4} + \text{rec\_func\_2}(5,5)$$

Потім при черговому виклику функції  $\text{rec\_func\_2}(5, 5)$  рекурсивні звернення припиняться і буде повернено значення  $\frac{1}{5}$ .

В результаті сформується така послідовність:

Значення  $\frac{1}{5}$  буде передано до  $\frac{1}{4} + \text{rec\_func\_2}(5,5)$  замість  $\text{rec\_func\_2}(5,5)$ , потім  $\frac{1}{4} + \frac{1}{5}$  до  $\frac{1}{3} + \text{rec\_func\_2}(5, 4)$  замість  $\text{rec\_func\_2}(5, 4)$  і т.д.

В результаті отримаємо ряд:

$$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5}$$

Ця послідовність операторів і дає результат обчислення суми:  
Сума = 2.28333333

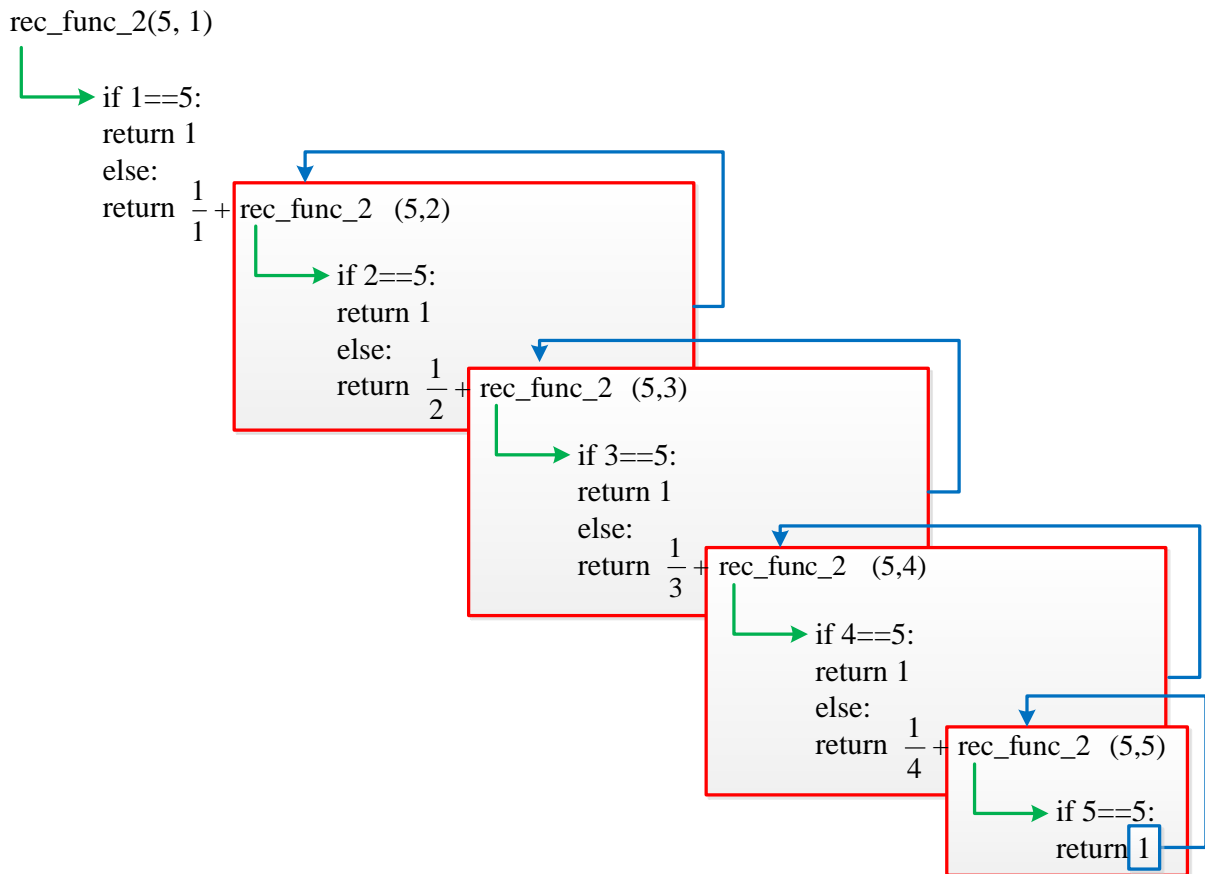


Рис.4.1. Графічне зображення роботи рекурсії

Ітерація і рекурсія засновані на керуючих структурах: ітерація використовує структуру повторення, рекурсія використовує структуру розгалуження.

```
def non_rec_func(n):
    S=0
    for i in range(1, n+1):
        S+=1/i
    return S

print(non_rec_func(5))
```

І ітерація, і рекурсія передбачають повторення: ітерація використовує структуру повторення явно, рекурсія – за допомогою повторних викликів функції.

Ітерація і рекурсія включають перевірку на завершення: ітерація завершується, коли перестає виконуватися умова продовження циклу, рекурсія завершується, коли розпізнається нерекурсивний випадок.

Як ітерація, так і рекурсія наближаються до завершення поступово.

Ітерація з її перевіркою повторення продовжує виконувати тіло циклу, поки умова продовження циклу не буде порушено. Рекурсія продовжує виробляти більш прості варіанти початкової задачі, поки не буде досягнутий нерекурсивний випадок.

І ітерація, і рекурсія може відбуватися нескінченно: ітерація потрапляє в нескінченний цикл, якщо умова продовження циклу ніколи не стає хибною; рекурсія триває нескінченно, якщо крок рекурсії не редукує задачу таким чином, що задача сходиться до нерекурсивного випадку.

Рекурсія має негативні сторони. Вона багато разів ініціалізує механізм виклику функції і збільшує пов'язані з ним витрати процесорного часу і пам'яті (кожне рекурсивне звернення створює копію її параметрів і локальних об'єктів). Ітерація зазвичай відбувається в межах функції, так що тут немає витрат на повторні виклики функції і додаткове виділення пам'яті. Налагодження рекурсивної функції викликає великі труднощі, ніж налагодження ітераційної функції.

Будь-яка проблема, яка може бути вирішена рекурсивно, може бути також вирішена і ітераційно (не рекурсивно).

Рекурсивний підхід краще ітераційного в тих випадках, коли рекурсія природніше відображає математичну сторону задачі і призводить до програми, яка простіше для розуміння.

Іншою причиною для вибору рекурсивного рішення є те, що ітераційне рішення може не бути очевидним.

### **4.3. Модульність в Python**

Якщо код програми є складним, розбиття її на окремі функції допомагає спростити його для візуального сприйняття. Якщо цього недостатньо, є сенс винести частину функцій та пов'язаних з ними оголошень за межі основного файлу програми.

Такі додаткові файли з кодом, що використовується в програмі, називаються модулями. Найчастіше вони містять оголошення функцій та констант, які далі можуть бути підключені (імпортовані) в головну програму і вільно в ній використовуватися. Об'єкти з модуля можуть бути імпортовані в

інші модулі. Файл утворюється шляхом додавання до імені модуля розширення `.py`. При імпорті модуля інтерпретатор шукає файл спочатку в поточному каталозі, потім в каталогах, зазначених у змінній оточення `PYTHONPATH`, потім в залежних від платформи шляхах за замовчуванням, а також в спеціальних файлах з розширенням `.pth`, які лежать в стандартних каталогах. Можна внести зміни в `PYTHONPATH` і в `.pth`, додавши туди свій шлях. Каталоги, в яких здійснюється пошук, можна подивитися в змінній `sys.path`.

Великі програми, як правило, складаються з стартового файлу – файлу верхнього рівня, і набору файлів-модулів. Головний файл займається контролем програми. У той же час модуль – це не тільки фізичний файл. Модуль являє собою колекцію компонентів. У цьому сенсі модуль – це простір імен, – *namespace*, і всі імена всередині модуля ще називаються атрибутами – такими, наприклад, як функції і змінні [5, 8, 9, 13].

Є велика кількість вбудованих модулів, що дозволяють виконувати складні математичні операції (`math`), працювати з датами (`datetime`)/часом (`time`), випадковими числами (`random`), операційною та файловою системами (`os`) та ін.

### ***Модуль math. Математичні функції***

Модуль `math` надає додаткові функції для роботи з числами, а також стандартні константи. Перш ніж використовувати модуль, необхідно підключити його за допомогою інструкції:

#### **`import math`**

Модуль `math` надає наступні стандартні константи:

**`pi`** – повертає число  $\pi$ .

**`e`** – повертає значення константи  $e$ .

```
import math
```

```
print(math.pi)
print(math.e)
```

Отримаємо:

```
3.141592653589793
```

```
2.718281828459045
```

Основні функції для роботи з числами:

*sin* (), *cos* (), *tan* () – стандартні тригонометричні функції (синус, косинус, тангенс). Значення вказується в радіанах;

*asin* (), *acos* (), *atan* () – зворотні тригонометричні функції (арксинус, арккосинус, арктангенс). Значення повертається в радіанах;

*degrees*() – перетворює радіани в градуси:

```
import math  
  
print(math.degrees(math.pi))
```

Отримаємо:

```
180.0
```

*radians*() – перетворює градуси в радіани:

```
import math  
  
print(math.radians(180.0))
```

Отримаємо:

```
3.1415926535897931
```

*exp*() – експонента;

*log*() – логарифм;

*sqrt*() – квадратний корінь:

```
import math  
  
print(math.sqrt(100), math.sqrt(25))
```

Отримаємо:

```
10.0, 5.0
```

*ceil*() – значення, округлене до найближчого більшого цілого:

```
import math  
  
print(math.ceil(5.49), math.ceil(5.50), math.ceil(5.51))
```



Отримаємо:

```
6.0, 6.0, 6.0
```

***floor()*** – значення, округлене до найближчого меншого цілого:

```
import math  
  
print(math.floor(5.49), math.floor(5.50), math.floor(5.51))
```

Отримаємо:

```
5.0, 5.0, 5.0
```

***pow(Число, Степень)*** – підносить Число до Степені:

```
import math  
  
print(math.pow(10, 2), 10 ** 2, math.pow(3, 3), 3 ** 3)
```

Отримаємо:

```
100.0, 100, 27.0, 27
```

***fabs()*** – абсолютне значення:

```
import math  
  
print(math.fabs(10), math.fabs(-10), math.fabs(-12.5))
```

Отримаємо:

```
10.0, 10.0, 12.5
```

***fmod()*** – остача від ділення:

```
import math  
  
print(math.fmod(10, 5), 10 % 5, math.fmod(10, 3), 10 % 3)
```

Отримаємо:

```
0.0, 0, 1.0, 1
```

***factorial()*** – факторіал числа:

```
import math  
print(math.factorial(5), math.factorial(6))
```

Отримаємо:

```
120, 720
```

### **Модуль *random*. Випадкові числа**

Більшість програм роблять одне і те ж при кожному виконанні, тому говорять, що такі програми визначені. Визначеність хороша річ до тих пір, поки ми вважаємо, що одні й ті ж обчислення повинні давати один і той же результат. Проте, в деяких програмах від комп'ютера потрібно непередбачуваність. Типовим прикладом є ігри, але є маса інших застосувань: зокрема, моделювання фізичних процесів або статистичні експерименти.

Змусити програму бути дійсно непередбачуваною завдання не таке просте, але є способи змусити її здаватися непередбачуваною. Одним з таких способів є генерування випадкових чисел і використання їх у програмі.

У Python є вбудований модуль, який дозволяє генерувати псевдовипадкові числа. З математичної точки зору, вони не істинно випадкові.

Модуль `random` дозволяє генерувати випадкові числа. Перш ніж використовувати модуль, необхідно підключити його за допомогою інструкції:

```
import random
```

Основні функції:

***random()*** – повертає псевдовипадкове дійсне число від 0.0 до 1.0:

```
import random

print(random.random())
print(random.random())
print(random.random())
```

Отримаємо:

```
0.42888905467511462
0.57809130113447038
0.20609823213950174
```

Числа, що видаються функцією *random()*, розподілені рівномірно – це означає, що всі значення рівноймовірні.

***uniform(start, end)*** – повертає псевдовипадкове дійсне число в діапазоні від *start* до *end*:

```
import random

print(random.uniform(0, 10))
print(random.uniform(0, 10))
```

Отримаємо:

```
1.6022955651881965
5.206693596399246
```

***randint(start, end)*** – повертає псевдовипадкове ціле число в діапазоні від *start* до *end*:

```
import random

print(random.randint(0, 10))
print(random.randint(0, 10))
```

Отримаємо:

```
10
6
```

***randrange(start, end, step)*** – повертає випадковий елемент з числової послідовності. Параметри аналогічні параметрам функції *range()*. Саме зі списку, що повертається функцією *range()*, і вибирається випадковий елемент:

```
import random

print(random.randrange(10))
print(random.randrange(0, 10))
print(random.randrange(0, 10, 2))
```

Отримаємо:

```
9
1
8
```

***choice(Послідовність)*** – повертає випадковий елемент з будь-якої послідовності (рядку, списку, кортежу):

```
import random

print(random.choice("string"))      # Випадковий символ з
                                    # рядку
print(random.choice(["s", "t", "r"])) # Випадковий елемент зі
                                    # списку
print(random.choice(("s", "t", "r"))) # Випадковий елемент з
                                    # кортежу
```

Отримаємо:

```
't'
's'
'r'
```

***shuffle(Список, Число від 0.0 до 1.0)*** – перемішує елементи списку випадковим чином. Функція перемішує сам список і нічого не повертає. Якщо другий параметр не вказано, то використовується значення, яке повернене функцією *random()*.

```
import random

lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
random.shuffle(lst)

print(lst)
```

Отримаємо:

```
[7, 1, 6, 10, 9, 4, 8, 3, 2, 5]
```

***sample(Послідовність, Кількість елементів)*** – повертає список із зазначеної кількості елементів. У цей список потраплять елементи з послідовності, вибрані випадковим чином. Як послідовність – можна вказати будь-який об'єкт, що підтримує ітерації.

```
import random

print(random.sample("string", 2) )
```

```
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(random.sample(lst, 2))

print(lst)                                # Сам список не змінюється

print(random.sample((1, 2, 3, 4, 5, 6, 7), 3) )
```

Отримаємо:

```
['s', 'g']
[8, 7]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[2, 4, 3]
```

### ***Імпорт з модулів та його види***

Для того щоб отримати доступ до функцій або змінних/констант з модуля та використати їх в основній програмі – його необхідно підключити до програми. Це можна зробити за допомогою інструкції **import МОДУЛЬ**, де модуль це ім'я іншого файлу Python без розширення *.py*.

```
import math
```

Дана команда імпортує модуль *math*. Тепер необхідно викликати з нього одну з функцій. Для того, щоб звернутися до змінної або функції з імпортованого модуля необхідно вказати його ім'я, поставити крапку і вказати необхідне ім'я **МОДУЛЬ.ФУНКЦІЯ/КОНСТАНТА**.

```
import math

print (math.e)
```

Отримаємо:

```
2.718281828459045
```

Запис *math.e* означає, що значення *e* знаходиться в просторі імен модуля *math*.

```
import math

print(math.sqrt(9))
```

Отримаємо:

```
3.0
```

Дізнатися, які функції і константи визначені в модулі можна за допомогою функції *dir()* :

```
import math

print(dir(math))
```

Отримаємо:

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'trunc']
>>>
```

В результаті виконання цієї команди інтерпретатор вивів всі імена, визначені в цьому модулі. У їх числі є і змінна *\_\_doc\_\_*, що дозволяє вивести опис модуля.

```
import math

print (math.__doc__)
```

Отримаємо:

```
This module is always available. It provides access to the
mathematical functions defined by the C standard.
```

*\_\_doc\_\_* є внутрішнім ім'ям рядка документації як змінної всередині функції.

```
import math

print (math.e.__doc__)
```

Отримаємо:

```
float(x) -> floating point number
```

```
Convert a string or number to a floating point number, if possible.
```

Крім того, дізнатися про функції, які містить модуль, можна через функцію *help()*:

```
import math

print(help(math))
```

Отримаємо:

```
Help on built-in module math:
NAME
  math
DESCRIPTION
  This module is always available. It provides access to the
  mathematical functions defined by the C standard.
FUNCTIONS
  acos(...)
    acos(x)

    Return the arc cosine (measured in radians) of x.
  ...
```

Якщо необхідно ознайомитися з описом конкретної функції модуля, то викликається довідка окремо для неї:

```
import math

print(help(math.sqrt))
```

Отримаємо:

```
Help on built-in function sqrt in module math:
sqrt(...)
  sqrt(x)

  Return the square root of x.
```

### *Імпорт окремої функції з модуля*

В Python можна імпортувати окрему функцію з модуля за допомогою наступної конструкції:

## **from МОДУЛЬ import ФУНКЦІЯ/КОНСТАНТА**

```
from math import sqrt
```

```
print(sqrt(9))
```

Отримаємо:

```
3.0
```

Таким чином, Python не створюватиме змінну *math*, а завантажить в пам'ять тільки функцію *sqrt()*. Тепер виклик функції можна робити, не звертаючись до імені модуля *math*.

Через кому можна перерахувати декілька функцій або змінних які необхідні.

```
from math import sqrt, factorial
```

```
print(sqrt(9))
```

```
print(factorial(9))
```

Отримаємо:

```
3.0
```

```
362880
```

Або вказати *\** і тоді можна звертатися до будь-яких функцій.

```
from math import *
```

```
print(sin(pi/2))
```

Отримаємо:

```
1.0
```

В якості параметра тригонометричні функції приймають значення кута в радіанах.

```
from math import *
```

```
print(help(sin))
```



Отримаємо:

```
Help on built-in function sin in module math:
```

```
sin(...)  
  sin(x)
```

```
Return the sine of x (measured in radians).
```

### ***Створення власних модулів***

Щоб створити власний модуль необхідно зберегти файл з власним ім'ям **ІМ'Я.py** (для модулів обов'язково вказується розширення *.py*), що містить якийсь код (вміст модуля). Наприклад створимо модуль назвавши його *my\_math*:

```
def my_func ():  
    print('test')
```

```
import my_math                # Імпорт модуля my_math  
  
my_math.my_func()            # Виклик функції my_func
```

Результатом запуску даного коду буде:

```
test
```

Якщо необхідно імпортувати одноіменні функції з кількох модулів, для них можна задати псевдоніми. Тоді функція буде теж скопійована в поточний простір імен, але під іншою назвою за допомогою наступної конструкції:

**from МОДУЛЬ import ФУНКЦІЯ/КОНСТАНТА as  
НОВЕ\_ІМ'Я**

Імпортування модуля виконує команди що містяться в ньому. Однак, повторне імпортування не приводить до виконання модуля, тобто він повторно не імпортується. Пояснюється це тим, що імпортування модулів в пам'ять – ресурсномісткий процес, тому зайвий раз Python його не виконує. Якщо було внесено зміни до модуля – необхідно його повторно імпортувати,

примусово вказати Python, що модуль вимагає повторного завантаження. Після виклику функції *reload()* із зазначенням в якості аргументу імені модуля, оновлений модуль завантажиться повторно.

```
import imp  
  
imp.reload(my_math)
```

Результатом запуску даного коду буде:

```
test  
<module 'mtest' from 'C:\\Python35-32\\mtest.py'>
```

### ***Каталоги пошуку модулів***

Для імпорту Python шукає файли, що зберігається в стандартному модулі *sys*, як змінну *path*. Можна отримати доступ до цього списку і змінити його (відрізняється для різних операційних систем).

```
import sys  
  
for place in sys.path: # path містить список шляхів пошуку  
    модулів  
    print(place)
```

Результатом запуску даного коду буде:

```
D:\Users\syad\AppData\Local\Programs\Python\Python35\Lib\idl  
elib  
D:\Users\syad\AppData\Local\Programs\Python\Python35\python  
35.zip  
D:\Users\syad\AppData\Local\Programs\Python\Python35\DLLs  
D:\Users\syad\AppData\Local\Programs\Python\Python35\lib  
D:\Users\syad\AppData\Local\Programs\Python\Python35  
D:\Users\syad\AppData\Local\Programs\Python\Python35\lib\site  
-packages
```

Список *sys.path* містить шляхи пошуку, одержувані з наступних джерел:

- шлях до поточного каталогу з виконуваним файлом;

– значення змінної оточення PYTHONPATH. Для додавання змінної в меню Пуск необхідно обрати пункт Панель керування (або Налаштування | Панель управління). Обрати пункт Система. Перейти на вкладку Додатково і натиснути кнопку Змінні середовища. У розділі Змінні середовища користувача натиснути кнопку Створити. В поле Ім'я змінної ввести "PYTHONPATH", а в полі Значення змінної задати шлях до папок, модулів через крапку з комою.

Після цих змін перезавантажувати комп'ютер не потрібно, достатньо заново запустити програму;

- шляхи пошуку стандартних модулів;
- вміст файлів з розширенням *pth*, розташованих в каталогах пошуку стандартних модулів, наприклад, в каталозі D:\Users\syad\AppData\Local\Programs\Python\Python35\lib\site-packages. Назва файлу може бути довільною, головне, щоб розширення файлу було *pth*. Кожен шлях (абсолютний або відносний) повинен бути розташований на окремому рядку.

Каталоги повинні існувати, в іншому випадку вони не будуть додані в список *sys.path*.

При пошуку модуля список *sys.path* проглядається зліва направо. Пошук припиняється після першого знайденого модуля. Таким чином, якщо в каталогах D:\Users\folder1 і D:\Users\folder2 існують однойменні модулі, то буде використовуватися модуль з папки D:\Users\folder1, оскільки він розташований першим у списку шляхів пошуку.

Список *sys.path* можна змінювати з програми за допомогою спискових методів. Наприклад, додати каталог в кінець списку можна за допомогою методу *append()*.

```
import sys

sys.path.append(r" D:\Users\folder1")      # Додаємо в кінець
списку
for place in sys.path:
    print(place)
```

Результатом запуску даного коду буде:

```
D:\Users\syad\AppData\Local\Programs\Python\Python35\Lib\idlib
D:\Users\syad\AppData\Local\Programs\Python\Python35\python35.zip
D:\Users\syad\AppData\Local\Programs\Python\Python35\DLLs
D:\Users\syad\AppData\Local\Programs\Python\Python35\lib
D:\Users\syad\AppData\Local\Programs\Python\Python35
D:\Users\syad\AppData\Local\Programs\Python\Python35\lib\site-packages
D:\Users\folder1
```

Додати каталог в початок списку – за допомогою методу *insert()*.

```
import sys

sys.path.insert(0, r" D:\Users\folder2")      # Додаємо на
початку списку
for place in sys.path:
    print(place)
```

Результатом запуску даного коду буде:

```
D:\Users\folder2

D:\Users\syad\AppData\Local\Programs\Python\Python35\Lib\idlib
D:\Users\syad\AppData\Local\Programs\Python\Python35\python35.zip
D:\Users\syad\AppData\Local\Programs\Python\Python35\DLLs
D:\Users\syad\AppData\Local\Programs\Python\Python35\lib
D:\Users\syad\AppData\Local\Programs\Python\Python35
D:\Users\syad\AppData\Local\Programs\Python\Python35\lib\site-packages
D:\Users\folder1
```

Символ *r* перед лапками дозволяє не інтерпретувати спеціальні послідовності. Якщо використовуються

звичайні рядки, то необхідно подвоїти кожен слеш в шляху:

```
sys.path.append ("D:\\Users\\folder1\\folder2\\folder3")
```

### ***Пакети***

Коли модулів стає забагато, виникає необхідність групувати їх далі. Для цього файли модулів розкладаються по папках.

Відомо, що інтерпретатор шукає модулі в поточній папці та у спеціально призначеному для цього місці, отже необхідно якимось чином показати йому, що папка поряд з вашою програмою – не просто папка з файлами, а містить модулі для підключення. Для цього в папці повинен знаходитися файл `__init__.py` – він може бути порожнім, але сама його наявність сигналізує інтерпретатору, що папка із ним є пакетом модулів і може використовуватися в програмі [5, 8, 9, 13].

***Пакетом*** називається каталог з модулями, в якому розташований файл ініціалізації `__init__.py`.

Як і модулі, пакети створюють нові простори імен:

```
import my_package.my_math
# Модуль my_math шукатиметься в пакеті my_package

print(my_package.my_math.exp(1))
```

Отримаємо:

```
2.718281828459045
```

Всі розглянуті види імпорту поширюється також на пакети. Лише в іменах додається додатковий елемент через крапку – назва пакету. Пакети можуть вкладатися в інші пакети, аналогічно додаючи нові простори імен.

Як і модулі, пакети можуть містити код, який буде виконано під час ініціалізації пакету, – він записується в самому файлі `__init__.py`

## Завдання на комп'ютерний практикум

1. Числа  $m$  та  $k$  ( $3 \leq k \leq 10$ ) вводяться з клавіатури. Згенерувати та вивести на екран  $m$  цілих (дійсних) випадкових чисел з проміжку, вказаному у пункті а. Виведення на екран здійснювати по  $k$  чисел у рядку.

2. Розробити програму, дотримуючись таких вимог:

- число  $n$  (кількість елементів списку) – іменована константа;
- елементи списку – псевдовипадкові числа, згенеровані на інтервалі  $[a, b]$ , де  $a$  і  $b$  вводяться з клавіатури ( $a < b$ );
- усі вхідні дані і також елементи списку виводяться на екран.

1	В одновимірному масиві (списку), що складається з $n$ дійсних елементів, обчислити: 1) суму від'ємних елементів; 2) добуток елементів списку, розташованих між максимальним і мінімальним елементами.
---	---

3. Розробити програму, дотримаючись таких вимог:

- розміри масиву  $n$  і  $m$  – ввести з клавіатури;
- елементи масиву – псевдовипадкові числа, згенеровані на інтервалі  $[a, b]$ , де  $a$  і  $b$  ( $a < b$ ) вводяться з клавіатури;
- усі вхідні та вихідні дані і також елементи початкової матриці та отриманої виводити на екран.

1	Реалізувати програму, яка міняє місцями перший і останній стовпці квадратної матриці.
---	---

## Запитання для самоконтролю

1. Яким чином можна згенерувати випадкове число?
2. Для чого існує функція `random()`?
3. Яким чином генеруються цілі випадкові числа на певному інтервалі?
4. Як згенерувати дійсні випадкові числа на певному інтервалі?
5. Що називають функцією?
6. Як відбувається звернення до функції?
7. Чи кожна функція повинна мати оператор повернення?
8. Що таке локальні змінні?
9. Що таке глобальні змінні?

10. Що таке фактичні параметри функції?
11. Що таке формальні параметри?
12. Чи можуть ідентифікатори фактичних і формальних параметрів співпадати?
13. Чи обов'язково кількість фактичних і формальних параметрів повинні співпадати?
14. Чи може глобальна змінна бути розташована у тілі програми?
15. Чи можна у середині однієї функції оголошувати іншу функцію?
16. Що таке документаційні рядки?

## РОЗДІЛ 5. РОБОТА З ФАЙЛАМИ

### 5.1. Уведення інформації у файли

Поки програма виконується її дані зберігаються в пам'яті (Random Access Memory (RAM)). Коли програма завершується або комп'ютер вимикають, дані з пам'яті зникають. Щоб зберігати дані постійно, необхідно помістити їх в файл. Файли зазвичай зберігають на жорсткому диску або носії.

Коли є велика кількість файлів, їх часто організують в директорії (папки). Кожен файл розпізнається за унікальним ім'ям або по комбінації імені файлу та імені директорії.

Читаючи і записуючи файли, програми можуть обмінюватися інформацією одна з одною і створювати придатні для друку формати, подібні PDF.

Робота з файлами в чомусь подібна роботі з книгами. Щоб прочитати книгу необхідно її відкрити. Коли прочитано необхідне – закрити. Поки книга відкрита, можна або її читати або в неї записувати. В обох випадках відомо, в якому місці книги знаходитесь. Більшу частину часу книга читається по порядку, але також можна пропустити якусь частину.

Все це також відноситься до файлів. Щоб відкрити файл необхідно вказати його ім'я і визначити необхідно його читати або записати в нього якусь інформацію.

Операція відкриття файлу створює файловий об'єкт.

**fileobj = open(filename, mode)**

- fileobj – це об'єкт файлу, що повертається функцією open ();
- filename – це рядок, що представляє собою ім'я файлу;
- mode – це рядок, який вказує на тип файлу і дії, які необхідно над ним зробити.

Перша літера рядка *mode* вказує на операцію (табл. 5.1):

Таблиця 5.1. Операції при роботі з файлами

Літера	Значення
r	Читання
w	Запис Якщо файлу не існує, він буде створений. Якщо файл існує, він буде перезаписаний



Продовження таблиці 5.1.

Літера	Значення
x	Запис, але тільки якщо файлу ще не існує
a	Додавання даних в кінець файлу, якщо він існує

Друга літера рядка *mode* вказує на тип файлу:

- t (або нічого) означає, що файл текстовий;
- b означає, що файл бінарний.

Після відкриття файлу викликаються функції для читання або запису даних.

І по завершенню потрібно закрити файл.

**Текстові файли** – файли які містять друковані символи і пробіли, згруповані в рядки, які відокремлені один від одного символами нового рядка. Оскільки Python спеціально створений для оброблення текстових файлів, він надає методи, які роблять таку роботу простою.

```
week_days = "Monday,
Tuesday,
Wednesday,
Thursday,
Friday,
Saturday,
Sunday"
print(len(week_days))
```

Отримаємо:

```
62
```

*write()* – записує звичайний рядок в файл. Якщо в якості параметра вказано *Unicode*-рядок, то проводиться спроба перетворити його в звичайний рядок. Так як за замовчуванням використовується кодування *ASCII*, спроба перетворити *Unicode*-рядок (що містить українські/російські літери) в звичайний рядок призведе до генерації винятку *UnicodeEncodeError*.

```
f = open('file.txt', 'wt')
f.write(week_days)
f.close()
```

Функція `write()` повертає число записаних байтів. Вона не додає ніяких пробілів або символів нового рядка, як це робить функція `print()`. За допомогою функції `print()` також можна записувати дані в текстовий файл:

```
f = open('file.txt', 'wt')
print(week_days, file=f)
f.close()
```

За замовчуванням функція `print()` додає пробіл після кожного аргументу і символ нового рядка в кінці. Для того щоб функція `print()` працювала як функція `write()`, треба передати їй два наступних аргументи:

- `sep` (роздільник, за замовчуванням це пробіл, ' ');
- `end` (символ кінця файлу, за замовчуванням це символ нового рядка, '\n').

Функція `print()` використовує значення за замовчуванням, якщо тільки не передати їй щось інше.

```
f = open('file.txt', 'wt')
print(week_days, file=f, sep=",", end="")
f.close()
```

## 5.2. Зчитування даних з файлу

Щоб повністю зчитати весь файл можна використати функцію `read()` без аргументів. Проте, потрібно мати на увазі, що файл розміром 1 Гбайт споживає 1 Гбайт пам'яті:

```
f = open('file.txt', 'rt')
days = f.read()
print(days)
f.close()
```

Результатом запуску даного коду буде:

```
Monday,
Tuesday,
Wednesday,
Thursday,
Friday,
Saturday,
```

## Sunday

Можна вказати максимальну кількість символів, яку функція `read()` поверне за один виклик.

```
f = open('file.txt', 'rt')
days = f.read(20)
print(days)
f.close()
```

Результатом запуску даного коду буде:

```
Monday,
Tuesday,
Wed
```

Після того як було зчитано весь файл, подальші виклики функції `read()` повертатимуть порожній рядок (").

Також можна зчитувати файл по одному рядку за раз за допомогою функції `readline()`.

```
f = open('file.txt', 'rt')
print(f.readline())
f.close()
```

Результатом запуску даного коду буде:

```
Monday,
```

Для текстового файлу навіть порожній рядок має довжину, рівну 1 (символ нового рядка), такий рядок буде вважатися `True`. Коли весь файл буде зчитано, функція `readline()` (як і функція `read()`) поверне порожній рядок, який буде розцінено як `False`.

Найпростіший спосіб зчитати текстовий файл – використати ітератор, що буде повертати по одному рядку за раз.

```
f = open('file.txt', 'rt')
for line in f:
    print(line)
f.close()
```

Результатом запуску даного коду буде:

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

### *Автоматичне закриття файлу*

Якщо забути закрити файл після роботи над ним, його закрий Python після того, як буде видалено останнє посилання на нього. Це означає, що, якщо відкрити файл і не закрити його явно, він буде закритий автоматично по завершенні функції. Але можна відкрити файл всередині довгої функції або навіть основного розділу програми. Файл повинен бути закритий, щоб всі операції, що залишилися та записи були завершені.

У Python є *менеджери контексту* для очищення об'єктів на зразок відкритих файлів. Можна використовувати конструкцію:

#### **with вираз as змінна**

```
with open('file.txt', 'wt') as f:  
    f.write(week_days)
```

Після того як блок коду, розташований під менеджером контексту (в цьому випадку один рядок), завершиться (або нормально, або шляхом генерації виключення), файл буде закритий автоматично.

## Завдання на комп'ютерний практикум

1. З клавіатури вводиться текстовий рядок. Розробити програму, яка реалізує вказані дії.

1	а) підраховує кількість слів, які мають непарну довжину; б) виводить на екран частоту входження кожної літери; в) видаляє текст, що розміщено в круглих дужках.
---	---

### Питання для самоконтролю

1. Робота з файлами в Python.
2. Відкриття файлу.
3. Закриття файлу.
4. Читання з файлу та запис у файл.
5. Додаткові дії для файлів.
6. Модуль path.
7. Опишіть наявні в Python функції роботи з файлами і методи файлових об'єктів.

## РОЗДІЛ 6. ВИНЯТКИ

### 6.1. Загальні поняття

**Винятки** – це сповіщення інтерпретатора, порушені в разі виникнення помилки в програмному коді або при настанні якої-небудь події. Якщо в коді не передбачено оброблення винятків, то програма переривається і виводиться повідомлення про помилку.

Нагадаємо, що існує три типи помилок в програмі:

**Синтаксичні** – це помилки в імені оператора або функції, невідповідність закриваючих та відкриваючих лапок і т.д. Тобто помилки в синтаксисі мови. Як правило, інтерпретатор попередить про наявність помилки, а програма не виконуватиметься зовсім. Приклад синтаксичної помилки:

```
print("Невідповідність відкритих та закритих лапок!")
```

Результатом запуску даного коду буде:

```
SyntaxError: EOL while scanning string literal
```

**Семантичні** – це помилки в логіці роботи програми, які можна виявити тільки за результатами роботи скрипта. Як правило, інтерпретатор не попереджає про наявність помилки. А програма буде виконуватися, оскільки не містить синтаксичних помилок. Такі помилки досить важко виявити і виправити.

**Помилки часу виконання** – це помилки, які виникають під час роботи скрипта. Причиною є події, які не передбачені програмістом. Класичним прикладом служить ділення на нуль:

```
def test (x, y):  
    return x/y  
  
print(test(4, 2))
```

Результатом запуску даного коду буде:

```
Traceback (most recent call last):  
  File "G:\КПИ\2017-2018\Иностранцы\lab2.py", line 4, in  
<module>  
    print(test(4, 0))
```

```
File "G:\lab.py", line 2, in test
    return x/y
ZeroDivisionError: division by zero
```

В мові Python винятки порушуються не тільки при помилці, але і як повідомлення про настання будь-яких подій. Наприклад, метод *index()* збуджує виняток *ValueError*, якщо шуканий фрагмент не входить в рядок:

```
>>> "String".index("S")
0

>>> "String".index("s")
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    "String".index("s")
ValueError: substring not found
```

## 6.2. Оброблення винятків

Для оброблення винятків призначена інструкція *try* [5, 8, 9, 13]. Формат інструкції:

```
try:
    <БЛОК, В ЯКОМУ ПЕРЕХОПЛЮЮТЬСЯ
    ВИНЯТКИ>
except <ВИНЯТОК_1> as <ОБ'ЄКТ ВИНЯТКУ>:
    <БЛОК, ЩО ВИКОНУЄТЬСЯ ПРИ ЗБУДЖЕННІ
    ВИНЯТКУ>
...
except <ВИНЯТОК_N> as <ОБ'ЄКТ ВИНЯТКУ>:
    <БЛОК, ЩО ВИКОНУЄТЬСЯ ПРИ ЗБУДЖЕННІ
    ВИНЯТКУ>
else:
    <БЛОК, ЩО ВИКОНУЄТЬСЯ, ЯКЩО ВИНЯТКУ НЕ
    ВИНИКЛО>
finally:
    < БЛОК, ЩО ВИКОНУЄТЬСЯ В БУДЬ-ЯКОМУ
    ВИПАДКУ>
```

Інструкції, в яких перехоплюються винятки, повинні бути розташовані всередині блоку *try*. У блоці *except* в параметрі <Виняток\_1> вказується клас оброблюваного винятку.

Наприклад, щоб обробити виняток, що виникає при діленні на нуль:

```
try:                                # Перехоплюється виняток
    x=1/0                            # Помилка: ділення на 0
except ZeroDivisionError:          # Вказуємо клас винятку
    print ("Обробили ділення на 0")
    x=0
    print(x)
```

Результатом запуску даного коду буде:

```
Обробили ділення на 0
0
```

Якщо в блоці *try* згенеровано виняток, то управління передається блоку *except*. У разі, якщо виключення не відповідає зазначеному класу, управління передається наступному блоку *except*. Якщо жоден блок *except* не відповідає винятку, то виняток "спливає" до обробника більш високого рівня. Якщо виняток ніде не обробляється в програмі, то управління передається обробнику за замовчуванням, який зупиняє виконання програми і виводить стандартну інформацію про помилку. Таким чином, в обробнику може бути кілька блоків *except* з різними класами винятків. Крім того, один обробник можна вкласти в інший.

```
try:                                # Обробляється виняток
    try:                             # Вкладений обробник
        x=1/0                        # Помилка: ділення на 0
    except NameError:
        print ("Невизначений ідентифікатор")
    except IndexError:
        print ("неіснуючий індекс")
        print ("Вираз після вкладеного обробника")
    except ZeroDivisionError:
        print ("Обробка ділення на 0")
        x=0
```



```
print(x)
```

Результатом запуску даного коду буде:

```
Оброблення ділення на 0  
0
```

У вкладеному обробнику не вказано виняток *ZeroDivisionError*, тому виняток "спливає" до обробника більш високого рівня. Після оброблення винятку управління передається інструкції, розташованій відразу після обробника.

В інструкції *except* можна вказати відразу кілька винятків, перерахувавши їх через кому всередині круглих дужок.

```
try:  
    x = 1/0  
except (NameError, IndexError, ZeroDivisionError):  
  
# Оброблення відразу декількох винятків  
  
    x = 0  
print (x)
```

Якщо в інструкції *except* не вказано клас винятку, то такий блок перехоплює всі винятки.

```
try:  
    x = 1/0  
except: # Оброблення всіх винятків  
    x = 0  
print (x)
```

На практиці слід уникати порожніх інструкцій *except*, оскільки можна перехопити виняток, яке є лише сигналом системі, а не помилкою.

Якщо в обробнику присутній блок *else*, то інструкції всередині цього блоку будуть виконані тільки при відсутності помилок. При необхідності виконати будь-які завершальні дії незалежно від того, було згенеровано виняток чи ні, слід скористатися блоком *finally*.

```
try:
    x = 10/2                # Немає помилки
    #x = 10/0              # Помилка: ділення на 0
except ZeroDivisionError:
    print ("Ділення на 0")
else:
    print ("Блок else")
finally:
    print ("Блок finally")
```

Результат виконання при відсутності винятку:

```
Блок else
Блок finally
```

Послідовність виконання блоків при наявності винятку буде інший:

```
Ділення на 0
Блок finally
```

При наявності винятку і відсутності блоку *except* інструкції всередині блоку *finally* будуть виконані, але виняток не буде оброблено. Він продовжить "спливання" до обробника більш високого рівня.

```
try:
    x = 10/0
finally:
    print ("Блок finally")
```

Якщо користувацький обробник відсутній, то управління передається обробнику за умовчанням, який перериває виконання програми і виводить повідомлення про помилку.

```
Блок finally
Traceback (most recent call last):
  File "G:\lab.py", line 2, in <module>
    x = 10/0
ZeroDivisionError: division by zero
```

### 6.3. Класи вбудованих винятків

Всі вбудовані виключення в мові Python представлені у вигляді класів. Ієрархія вбудованих класів винятків:

```
BaseException
  GeneratorExit
  KeyboardInterrupt
  SystemExit
  Exception
    StopIteration
    Warning
      BytesWarning, ResourceWarning,
      DeprecationWarning, FutureWarning, ImportWarning,
      PendingDeprecationWarning, RuntimeWarning,
SyntaxWarning,
  UnicodeWarning, UserWarning
  ArithmeticError
    FloatingPointError, OverflowError, ZeroDivisionError
  AssertionError
  AttributeError
  BufferError
  EnvironmentError
    IOError
    OSError
      WindowsError
  EOFError
  ImportError
  LookupError
    IndexError, KeyError
  MemoryError
  NameError
    UnboundLocalError
  ReferenceError
  RuntimeError
    NotImplementedError
  SyntaxError
    IndentationError
```

```
TabError
SystemError
TypeError
ValueError
UnicodeError
UnicodeDecodeError, UnicodeEncodeError
UnicodeTranslateError
```

Основна перевага використання класів для оброблення винятків полягає в можливості вказівки базового класу для перехоплення всіх винятків відповідних класів-нащадків. Наприклад, для перехоплення ділення на нуль було використано клас *ZeroDivisionError*. Якщо замість цього класу вказати базовий клас *ArithmeticError*, то перехоплюватимуться винятки класів *FloatingPointError*, *OverflowError* і *ZeroDivisionError*.

```
try:
    x = 1/0
except ArithmeticError:
    print ("Обробили ділення на 0")
# Вказано базовий клас
```

### Запитання для самоконтролю

1. Поняття помилки.
2. Що таке виняткові ситуації і яким чином здійснюється їх оброблення у Python?
3. Блоки try – except.
4. Атрибути винятків, ініціювання винятків.
5. Для чого використовується гілка finally в інструкції try?
6. Чи можна гілку finally поєднувати з гілками except?
7. Які два класи виняткових ситуацій наявні в Python?
8. Який із класів виняткових ситуацій рекомендується використовувати у програмах?

## Завдання до розрахункової роботи

### **Вимоги до програмного коду:**

Модуль має містити дві функції (рекурсивний та ітераційний алгоритми розв'язання задачі).

Значення  $x$  та точність  $\varepsilon$  вводяться з клавіатури.

Виконати перевірку зі значенням стандартної функції.

Мінімально використовувати вбудовані методи і функції.

Сам модуль та всі функції повинні містити документаційні рядки.

**Задані:** значення  $x$ , точність  $\varepsilon$ . Скласти підпрограму розрахунку функції  $y$  з точністю  $\varepsilon$ , використовуючи рекурсивний та ітераційний алгоритми розв'язання задачі.

Визначити, яку кількість членів ряду необхідно підсумувати для досягнення зазначеної точності (порівняти результат підсумовування зі значенням стандартної функції).

Варіант	$f(x)$	$y$	Діапазон аргументу
1	$\text{arctg}(x)$	$x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots + (-1)^n \frac{x^{2*n+1}}{2*n+1}$	$ x  < 1$

## ПРЕДМЕТНИЙ ПОКАЖЧИК

*Акумулятор*, 9

*Алгоритм*, 17, 19 - 27, 31, 91, 92, 97

*Альтернативна гілка*, 95, 102

*Анонімні функції*, 128

*Аргументи*, 40, 61, 62, 68, 70, 76, 78, 82, 104, 117, 119 - 121, 123 - 130, 143, 154, 155

*Арифметико-логічний пристрій*, 8, 9

*Архітектура комп'ютера*, 8  
*Асемблер*, 27, 28

*Базис*, 11

*Блок-схема*, 21, 22, 91 - 96

*Вага розряду*, 11, 12

*Виведення даних*, 40

*Видалення елемента*, 75, 87

*Виконуваність операцій*, 20

*Вилучення підрядку*, 59

*Винятки*, 30, 46, 49, 59, 67, 72, 74, 84, 86, 100, 127, 154, 160 - 165

*Випадкові числа*, 134, 136, 137, 138

*Вирівнювання*, 62, 63, 66, 67, 69, 70

*Вкладені списки*, 107

*Вкладені цикли*, 108, 109

*Включення*, 106, 107

*Властивості алгоритмів*, 19, 20

*Внутрішні функції*, 128

*Глобальна змінна*, 121, 122

*Двійкова система числення*, 13

*Денормалізована форма*, 18

*Десяткова система числення*, 12

*Детермінованість алгоритму*, 20

*Динамічна типізація*, 42, 43

*Дискретність інформації*, 20

*Дискретність роботи алгоритму*, 20

*Документаційні рядки*, 126

*Електронна обчислювальна машина*, 8

*Жорсткий диск*, 9

*Заміна символів*, 70

*Звернення до елемента*, 72, 84,

*Звернення до символу*, 58

*Змінна*, 41, 42, 43

*Змішана позиційна система числення*, 11

*Зсув*, 19, 58, 59, 72, 73, 76, 77

*Інструментальне програмне забезпечення*, 26

*Ідентифікатор*, 37, 118

*Іменовані аргументи*, 123, 124, 125

*Імпорт*, 133, 139, 141, 143, 145

*Інтерпретатор*, 28 – 31, 35 – 37, 43, 83, 85, 98, 127, 133, 140, 145

Ітерація, 132, 133

*Керуючі символи*, 56

Ключ, 82 - 88, 125

Коментар, 23, 24, 26, 39, 126

Компілятор, 28, 29, 30

Комплексні числа, 44

Конструкція `while`, 98

Копіювання, 78, 88

Кортеж, 52, 61, 66, 80, 81, 84, 88, 103, 104, 124, 125

*Логічний вираз*, 93, 98

Логічний оператор, 52, 53

Локальна змінна, 121

*Мантиса*, 18

Масив, 70

Масовість алгоритму, 21

Машинний код, 27, 28, 29

Машинний порядок, 19

Менеджер контексту, 156, 157

Методи рядків, 61,

Методи словників, 86

Методи списків, 73,

Мови програмування, 21, 27, 28, 31

*Налагодження*, 29, 30

Налагоджувачі, 26, 27

Непозиційна система числення, 11

Нескінченні цикли, 100

Номер розряду, 12

Нормалізована форма, 19

*Об'єднання рядків*, 58, 62

Об'єкт, 43

Оброблення винятків, 160, 161

Однорідна позиційна система числення, 12

Оператор, 44, 47, 52

Оператор `pass`, 97, 118

Оператори присвоєння, 47

Операція, 44, 45

Основа системи числення, 11, 18

*Пакети*, 145

Пам'ять, 9

Параметри функції, 83, 118

Переведення цілого числа, 13, 35

Перетворення типів, 48, 57

Подання даних, 10

Подання цілих від'ємних чисел, 16

Подання цілих додатніх чисел, 15

Позиційна система числення, 11, 12

Позиційні аргументи, 123, 124, 125

Помилки виконання, 30

Потік виконання, 95, 96, 99, 102, 127

Представлення дійсних чисел, 17, 19

Прикладне програмне забезпечення, 26, 27

Пристрій управління, 8, 9

Пріоритет операцій, 48, 51

Програма, 26, 27

Програмне забезпечення,

Програмування, 29

**Редактори**, 27, 37  
**Регістр**, 37, 63  
**Рекурсія**, 129, 132, 133  
**Розгалуження**, 24, 50, 91, 94, 97, 134  
**Розряд**, 11 - 17

**Семантичні помилки**, 30, 10, 160  
**Сильна типізація**, 43  
**Синтаксичні помилки**, 30, 160  
**Система числення**, 10 - 13  
**Системне програмне забезпечення**, 26  
**Скінченність алгоритму**, 20  
**Слабка типізація**, 43  
**Слідування**, 91  
**Словники**, 82, 85, 86  
**Сортування**, 77, 78  
**Список**, 62  
**Спрямованість алгоритму**, 21  
**Статична типізація**, 42, 43

**Текстові файли**, 154  
**Тестування**, 29  
**Тетрада**, 14, 15

**Тип даних**, 42, 43, 46, 50, 64, 71  
**Типізація**, 42, 43  
**Тріада**, 14, 15  
**Транслятори**, 26 – 28, 30

**Уведення даних**, 22, 40, 98

**Фактичні параметри**, 119  
**Формальні параметри**, 118  
**Форматування рядків**, 63, 64  
**Функція**, 40, 49, 60, 61, 63, 74, 77 – 79, 84, 88, 94, 104, 105, 108, 117

**Цикл**, 22, 91, 92, 97 - 104  
**Цикл for**, 98, 102, 103, 109, 110  
**Цілі числа**, 15, 41, 43, 44, 49, 78, 101, 115

**Числа з плаваючою точкою**, 43, 44, 78  
**Числові послідовності**, 104, 137

**Шістнадцяткова система числення**, 13, 27



## СПИСОК ЛІТЕРАТУРИ

1. Дунець Р. Б. Арифметичні основи комп'ютерної техніки / Р. Б. Дунець, О. Т. Кудрявцев. – Львів : Ліга-Прес, 2006. – 142 с.
2. Довгалець С. М. Алгоритмічні мови та програмування. Частина 1. Основи інформатики та комп'ютерної техніки. Навчальний посібник / С. М. Довгалець, Р. В. Маслій. – Вінниця : ВНТУ, 2009. – 116 с.
3. Кравчук А. И. Сборник лабораторных работ с примерами решения задач по алгоритмизации и программированию на языке Си : Учебно-методическое пособие для студентов высших технических учебных заведений / А. И. Кравчук, А. С. Кравчук. – Мн. : Технопринт, 2002. – 111 с.
4. Яковенко А. Основи програмування: методичні вказівки до виконання комп'ютерних практикумів з дисципліни "Основи програмування". Основи програмування мовою Python / А. В. Яковенко. – Київ : НТУУ "КПІ ім. І. Сікорського", 2017. – 87 с.
5. Язык программирования Python / Г. Россум, Ф. Л. Дж. Дрейк, Д. С. Откидач та ін. 2001. – 454 с.
6. Бизли Д. Python. Подробный справочник / Д. Бизли. – СПб. : Символ-Плюс, 2010. – 864 с.
7. Хахаев И. А. Python. Практикум по алгоритмизации и программированию на Python / И. А. Хахаев. – М. : Альт Линукс, 2010. – 126 с.
8. Любанович Б. Python. Простой Python. Современный стиль программирования / Б. Любанович. – СПб. : Питер, 2016. – 480 с.
9. Федоров Д. Ю. Основы программирования на примере языка Python : учеб. пособие / Д. Ю. Федоров. – СПб. : Питер, 2016. – 176 с.
10. Лутц М. Изучаем Python, 4-е издание / М. Лутц. – СПб. : Символ-Плюс, 2011. – 1280 с.
11. Доусон М. Програмуємо на Python / М. Доусон. – СПб. : Питер, 2014. – 416 с.

12. Мусин Д. Самоучитель Python. Выпуск 0.2 / Д. Мусин. – Pythonworld.ru, 2015. – 136 с.
13. Прохоренок Н. А. Python 3 и PyQt. Разработка приложений / Н. А. Прохоренок. – СПб. : БХВ-Петербург, 2012. – 704 с.
14. Кнут Д. Искусство программирования, том 1. Основные алгоритмы – The Art of Computer Programming, vol.1. Fundamental Algorithms / Д. Кнут. – М. : "Вильямс", 2006. – 720 с.

## Додаток 1. Функції та методи рядків

Функція або метод	Призначення	Приклад	Результат
<code>S = 'str'; S = "str"; S = '''str'''; S = ""str""</code>	Літерали рядків	<code>&gt;&gt;&gt; S1='33 ' &gt;&gt;&gt; S2="cows and a glass of fresh milk"</code>	
<code>S = "s\np\ta\nbbb"</code>	Екрановані послідовності	<code>&gt;&gt;&gt; S = 's\np\ta\nbbb' &gt;&gt;&gt; print(S)</code>	s p a bbb
<code>S = r"C:\temp\new"</code>	Неформатовані рядки (пригнічують екранування)	<code>&gt;&gt;&gt; S = r'C:\temp\new'</code>	C:\temp\new
<code>S = b"byte"</code>	Рядок байтів		
<code>S1 + S2</code>	Конкатенація (додавання рядків)	<code>&gt;&gt;&gt; S=S1+S2</code>	'33 cows and a glass of fresh milk'
<code>S1 * 3</code>	Повторення рядка	<code>&gt;&gt;&gt; S1*3</code>	'33 33 33 '
<code>S[i]</code>	Звернення за індексом	<code>&gt;&gt;&gt; S[5]</code>	'w'
<code>S[i:j:step]</code>	Витяг зрізу	<code>&gt;&gt;&gt; S[3:10:3]</code>	'csn'
<code>len(S)</code>	Довжина рядка	<code>&gt;&gt;&gt; len(S)</code>	33
<code>S.find(str, [start],[end])</code>	Пошук підрядка в рядку. Повертає номер першого входження або -1	<code>&gt;&gt;&gt; S.find('milk', 10, 30) &gt;&gt;&gt; S.find('milk', 10, 33)</code>	-1  29
<code>S.rfind(str, [start],[end])</code>	Пошук підрядка в рядку. Повертає номер останнього входження або -1	<code>&gt;&gt;&gt; S.rfind('s', 10, 33)</code>	26
<code>S.index(str, [start],[end])</code>	Пошук підрядка в рядку. Повертає	<code>&gt;&gt;&gt; S.index('s', 0,</code>	6

Функція або метод	Призначення	Приклад	Результат
	номер першого входження або викликає ValueError	32) >>> S.index('s')	6
S.rindex(str, [start],[end])	Пошук підрядка в рядку. Повертає номер останнього входження або викликає ValueError	>>> S.rindex('s', 0, 32)	26
S.replace(шаблон, заміна)	Заміна шаблону	>>> S.replace('fresh', 'warm')	'33 cows and a glass of warm milk'
S.split(символ)	Розбиття рядка по роздільнику	>>> L = S.split('and') >>> L	['33 cows ', 'a glass of fresh milk']
S.isdigit()	Чи складається рядок з цифр	>>> S.isdigit()	False
S.isalpha()	Чи складається рядок з букв	>>> S.isalpha()	False
S.isalnum()	Чи складається рядок з цифр або букв	>>> S.isalnum()	False
S.islower()	Чи складається рядок із символів в нижньому регістрі	>>> S.islower()	True
S.isupper()	Чи складається рядок із символів у верхньому регістрі	>>> S.isupper()	False
S.isspace()	Чи складається рядок з символів, що не відображаються (пробіл, символ переводу сторінки ('\f'), "новий рядок" ('\n'), "перевод каретки" ('\r'),	>>> S.isspace()	False

Функція або метод	Призначення	Приклад	Результат
	"горизонтальна табуляція" (\t) і "вертикальна табуляція" (\v))		
S.istitle()	Чи починаються слова в рядку з великої літери	>>> S.istitle()	False
S.upper()	Перетворення рядка до верхнього регістру	>>> S.upper()	'33 COWS AND A GLASS OF FRESH MILK'
S.lower()	Перетворення рядка до нижнього регістру	>>> S.lower() '33 cows and a glass of fresh milk'	
S.startswith(str)	Чи починається рядок S з шаблону str	>>> S.startswith('33') True	
S.endswith(str)	Чи закінчується рядок S шаблоном str	>>> S.endswith('33') False	
S.join(список)	Збірка рядка зі списку з роздільником S	>>> 'and'.join(L) '33 cows and a glass of fresh milk'	
ord(символ)	Символ в його код ASCII	>>> ord('m') 109 >>> ord('M') 77	
chr(число)	Код ASCII в символ	>>> chr(33) '!'	

Функція або метод	Призначення	Приклад	Результат
S.capitalize()	Перекладає перший символ рядка в верхній регістр, а всі інші в нижній	>>> S.capitalize() '33 cows and a glass of fresh milk'	
S.center(width, [fill])	Повертає відцентрований рядок, по краях якого стоїть символ fill (пробіл по замовчуванню)	>>> S.center(60, '@') '@@@@@ @@@@@ @33 cows and a glass of fresh milk@@@@ @@@@@ @@@@@'	
S.count(str, [start],[end])	Повертає кількість входжень підрядка, що не перетинаються, в діапазоні [початок, кінець] (0 і довжина рядка по замовчуванню)	>>> S.count('3', 0, 32) 2	
S.expandtabs([tabsize])	Повертає копію рядка, в якому всі символи табуляції замінюються одним або декількома пропусками, в залежності від поточного стовпця. Якщо TabSize не вказано, розмір табуляції покладається рівним	>>> S.expandtabs() '33 cows and a glass of fresh milk'	

Функція або метод	Призначення	Приклад	Результат
	8 пробілів		
S.lstrip([chars])	Видалення символів пробілів на початку рядка		
S.rstrip([chars])	Видалення символів пробілів в кінці рядка		
S.strip([chars])	Видалення символів пробілів на початку і в кінці рядка		
S.partition(шаблон)	Повертає кортеж, що містить частину перед першим шаблоном, сам шаблон, і частину після шаблону. Якщо шаблон не знайдено, повертається кортеж, що містить сам рядок, а потім два порожніх рядки	>>> S.partition('cows') ( '33', 'cows', ' and a glass of fresh milk')	
S.rpartition(sep)	Повертає кортеж, що містить частину перед останнім шаблоном, сам шаблон, і частину після шаблону. Якщо шаблон не знайдений, повертається кортеж, що містить два порожніх рядки, а потім сам рядок	>>> S.rpartition('dogs') ( '33 cows and a glass of fresh milk', ' ', '')	
S.swapcase()	Перекладає символи нижнього регістра в верхній, а верхнього – в нижній	>>> S.swapcase() '33 COWS AND A	

Функція або метод	Призначення	Приклад	Результат
		GLASS OF FRESH MILK'	
S.title()	Першу букву кожного слова переводить в великі букви, а всі інші в нижній регістр	>>> S.title() '33 Cows And A Glass Of Fresh Milk'	
S.zfill(width)	Робить довжину рядка не меншою width, по-необхідності заповнюючи перші символи нулями	>>> S.zfill(60) '000000000000 000000000000 0000033 cows and a glass of fresh milk'	
S.ljust(width, fillchar=" ")	Робить довжину рядка не меншою width, по-необхідності заповнюючи останні символи символом fillchar	>>> S.ljust(60, '*') '33 cows and a glass of fresh milk***** ***** *****'	
S.rjust(width, fillchar=" ")	Робить довжину рядка не меншою width, по-необхідності заповнюючи перші символи символом fillchar	>>> S.rjust(60, '*') '***** ***** *****33 cows and a glass of fresh milk'	
S.format(*args, **kwargs)	Форматування рядка		



## Додаток 2. Методи списків

Метод	Призначення	Приклад	Результат
list.append(x)	Додає елемент в кінець списку	>>> l = [1, 2, 3] >>> x = 4 >>> l.append(x)	[1, 2, 3, 4]
list.extend(L)	Розширює список list, додаючи в кінець усі елементи списку L	>>> L = [5, 6] >>> l.extend(L)	[1, 2, 3, 4, 5, 6]
list.insert(i, x)	Вставляє на i-ий елемент значення x	>>> l.insert(0, 2)	[2, 1, 2, 3, 4, 5, 6]
list.remove(x)	Видаляє перший елемент в списку, що має значення x	>>> l.remove(3)	[2, 1, 2, 4, 5, 6]
list.pop([i])	Видаляє i-ий елемент та повертає його значення. Якщо індекс не вказано, видаляється останній елемент	>>> l.pop(3)	4 >>> l [2, 1, 2, 5, 6]
list.index(x, [start [, end]])	Повертає положення першого елемента від start до end із значенням x	>>> l.index(2, 2, 4)	2
list.count(x)	Повертає кількість елементів зі значенням x	>>> l.count(2)	2
list.sort([key = функція])	Сортує список на основі функції	>>> l.sort()	[1, 2, 2, 5, 6]
list.reverse()	Розвертає список	>>> l.reverse()	[6, 5, 2, 2, 1]
list.copy()	Поверхнева копія списку	>>> l.copy()	
list.clear()	Очищення списку	>>> l.clear()	[]

### Додаток 3. Методи словників

Метод	Призначення	Приклад	Результат
dict.clear()	Очищує словник	<pre>&gt;&gt;&gt; dict_1 = {"a": 1, "b": 2} &gt;&gt;&gt; dict_1.clear()</pre>	{}
dict.copy()	Повертає копію словника	<pre>&gt;&gt;&gt; dict_2 = {"a": 1, "b": 2} &gt;&gt;&gt; dict_copy = dict_2</pre>	{'a': 1, 'c': 3, 'b': 2}
dict.get(key[, default])	Повертає значення ключа, але якщо він відсутній, то повертає default (по замовчуванню None)	<pre>&gt;&gt;&gt; dict_2.get('2', 'Not a dict')</pre>	'Not a dict'
dict.items()	Повертає пари (ключ, значення)	<pre>&gt;&gt;&gt; dict_2.items( )</pre>	<pre>&gt;&gt;&gt; list(dict_2.it ems()) [('a', 1), ('b', 2)]</pre>
dict.keys()	Повертає ключі в словнику	<pre>&gt;&gt;&gt; dict_2.keys()</pre>	<pre>&gt;&gt;&gt; list(dict_2.k eys()) ['a', 'b']</pre>
dict.pop(key[, default])	Видаляє ключ і повертає значення. Якщо ключ відсутній, повертає default (по замовчуванню генерує виняток)	<pre>&gt;&gt;&gt; dict_2.pop('c' , 'Not a dict')</pre>	'Not a dict'
dict.popitem()	Видаляє і повертає пару (ключ, значення). Якщо словник	<pre>&gt;&gt;&gt; dict_2.popite m()</pre>	('a', 1)

Метод	Призначення	Приклад	Результат
	порожній, генерує виняток <code>KeyError</code> .		
<code>dict.update([other])</code>	Поновлює словник, додаючи пари (ключ, значення) з <code>other</code> . Існуючі ключі перезаписуються. Повертає <code>None</code> (не новий словник)	<pre>&gt;&gt;&gt; dict_2.update(c=3, d=4)</pre>	<pre>{'d': 4, 'c': 3, 'b': 2}</pre>
<code>dict.values()</code>	Повертає значення словнику	<pre>&gt;&gt;&gt; dict_2.values()</pre>	<pre>&gt;&gt;&gt; list(dict_2.values()) [4, 3, 2]</pre>